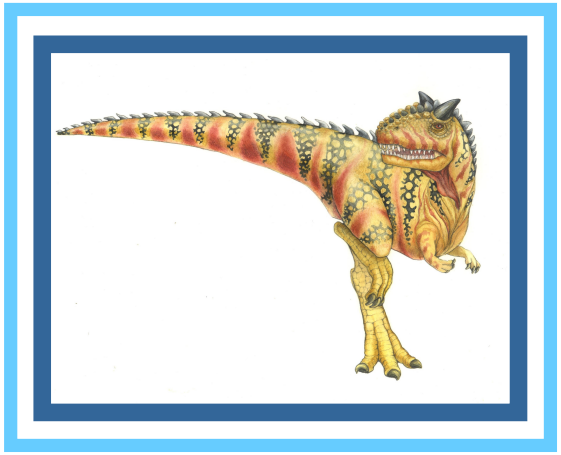
Chapter 9: Main Memory



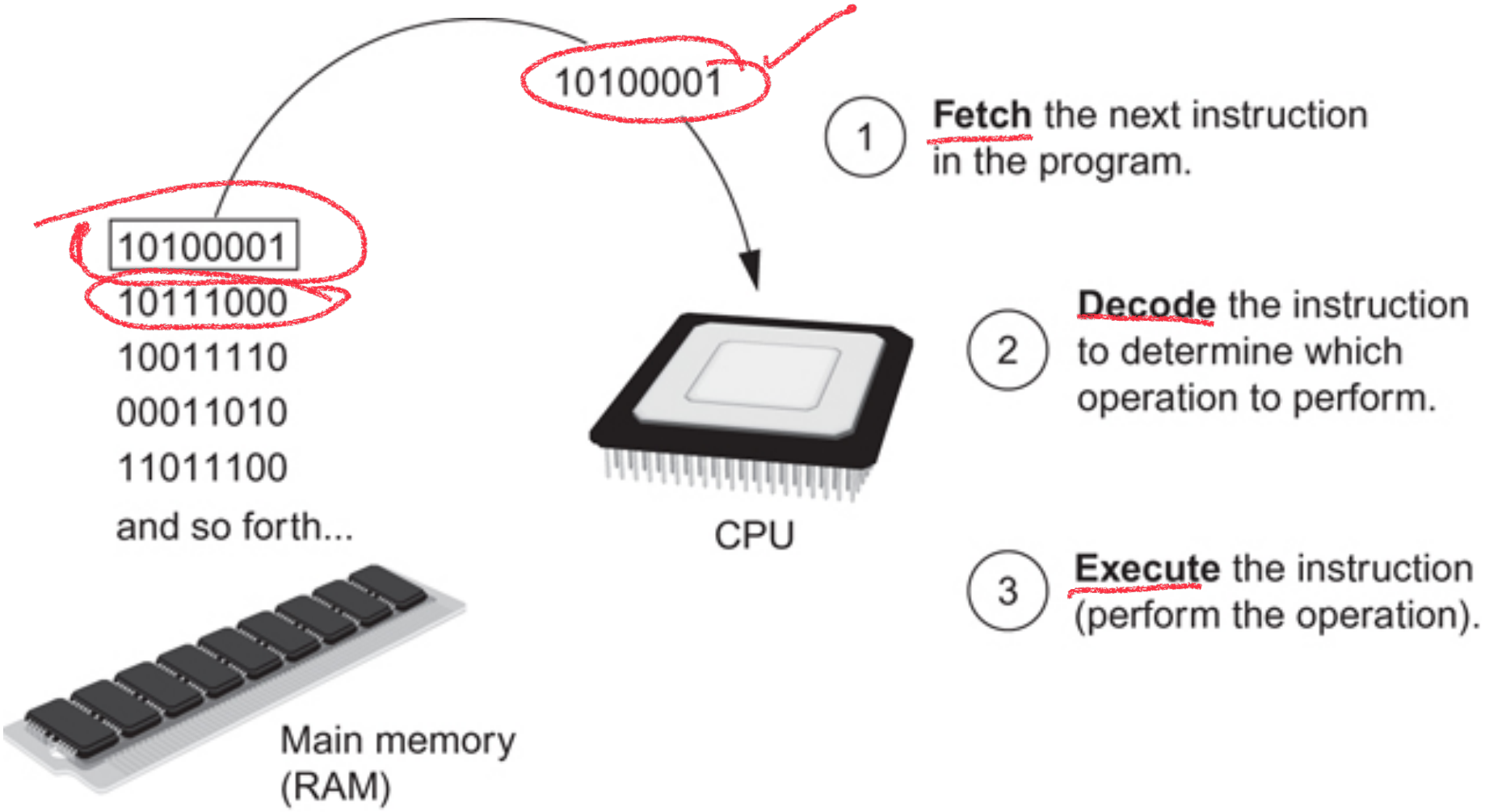
Objectives

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

Background

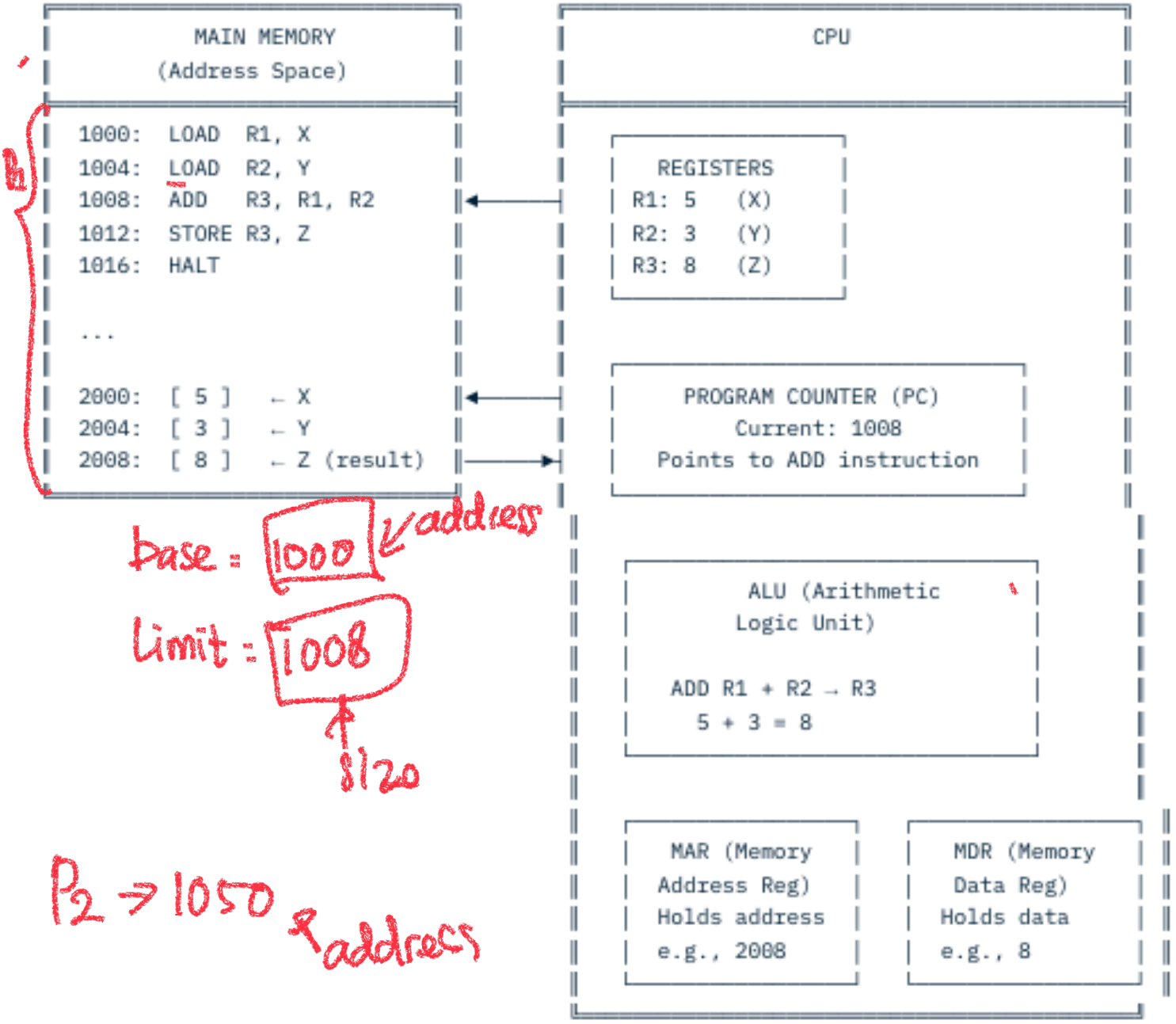
- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Memory unit only sees a stream of addresses + read requests, or address + data and write requests
- Register access in one CPU clock (or less)
  - Main memory can take many cycles, causing a **stall**
  - **Cache** sits between main memory and CPU registers

Recall : How a Program Works



CPU relies on the program counter to execute the next instruction

Simple Program: Z = X + Y



base = 1000 ← address

limit = 1008

$1120

P_2 → 1050 ← address

P_2 → base 2009

limit 1008

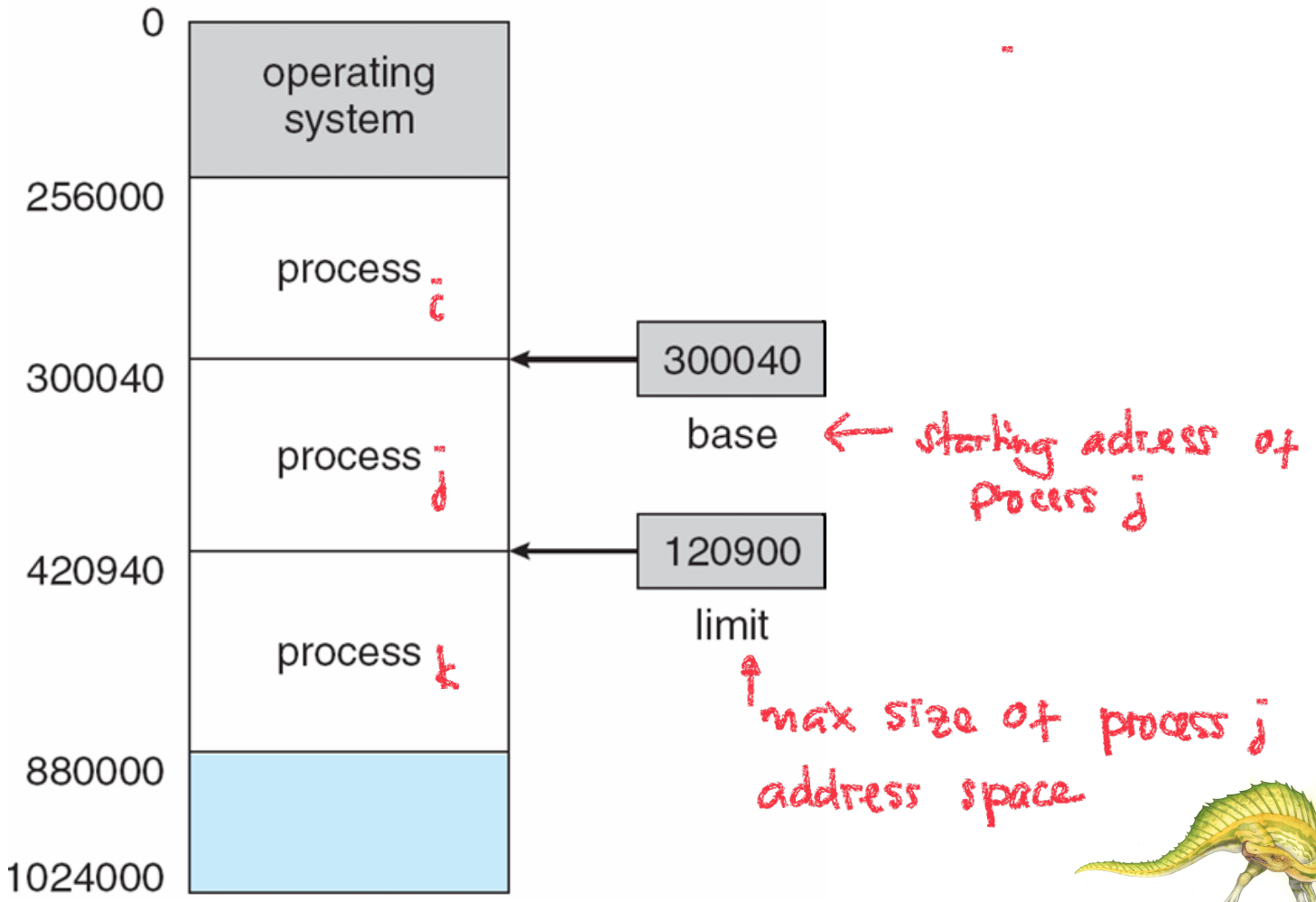
Background

- Protection of memory required to ensure correct operation
  - Make sure that each process has a separate memory space
    - fundamental to having multiple processes loaded in memory for concurrent execution

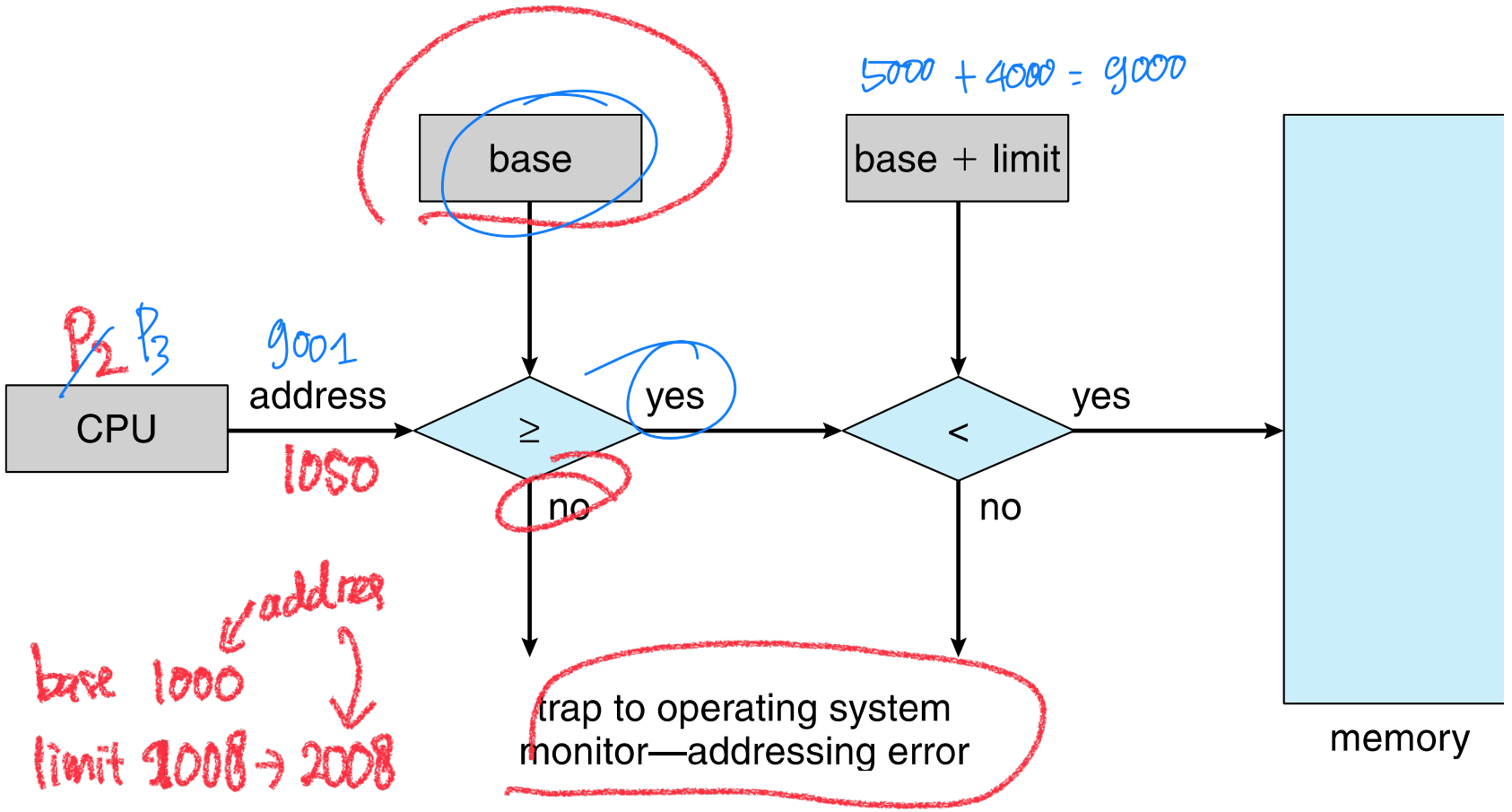
main objective of OS : memory management

Base and Limit Registers

- A pair of **base** and **limit registers** define the logical address space
- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user



Hardware Address Protection

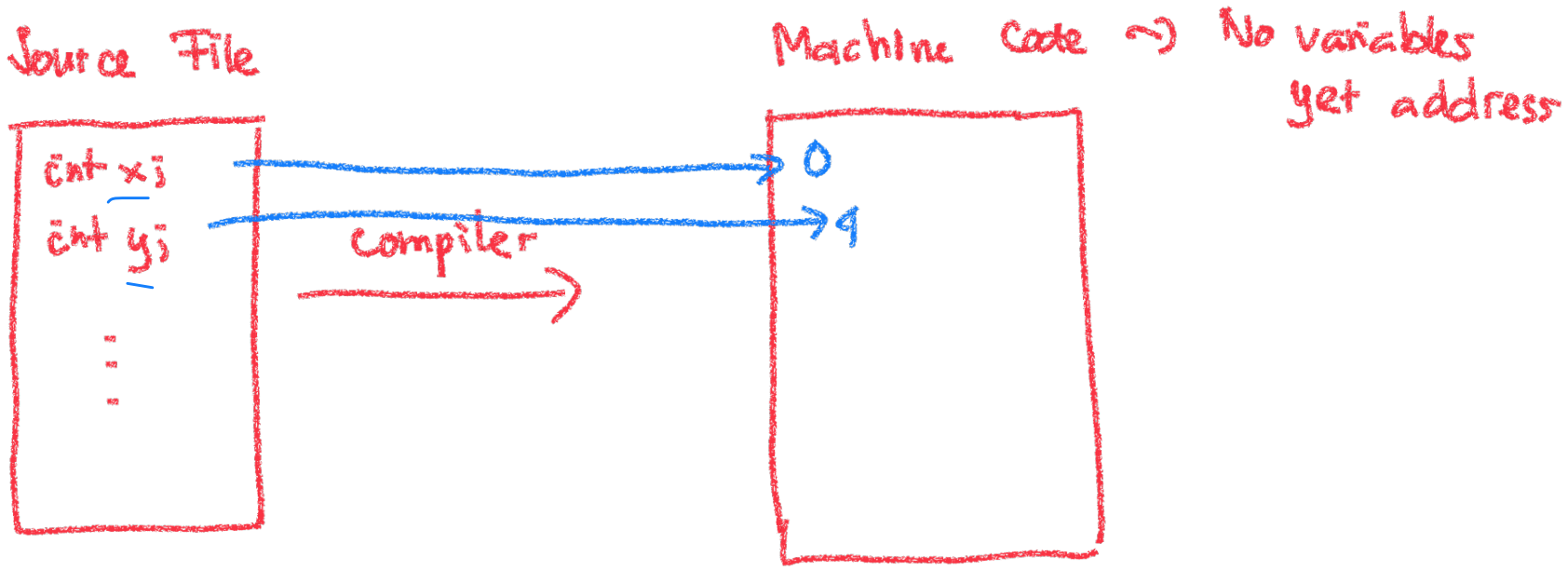


Address Binding (1)

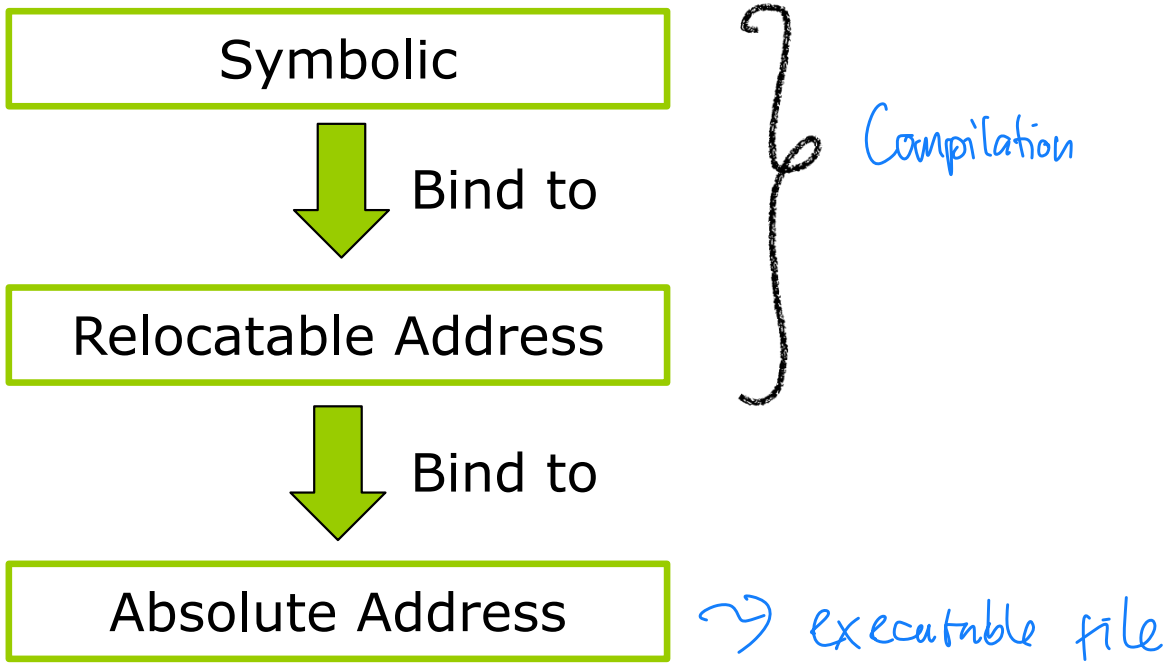
- Programs on disk, ready to be brought into memory to execute form an **input queue**
  - Without support, must be loaded into address 0000
  - Inconvenient to have first user process physical address always at 0000
  - How can it not be?

Address Binding (2)

- Further, addresses represented in different ways at different stages of a program's life
  - Source code addresses usually **symbolic** (e.g, a variable might be referred to by its name (`x` or `total`) rather than an exact memory address)
  - Compiled code addresses **bind** to **relocatable addresses**
    - the compiler translates symbolic addresses into *relocatable addresses*.
    - i.e. "14 bytes from beginning of this module"
  - Linker or loader will bind **relocatable addresses** to **absolute addresses**
    - i.e. 74014
  - Each binding maps one address space to another


Source File
int x;
int y;
compiler
Machine Code ~) No variables yet address
0
4

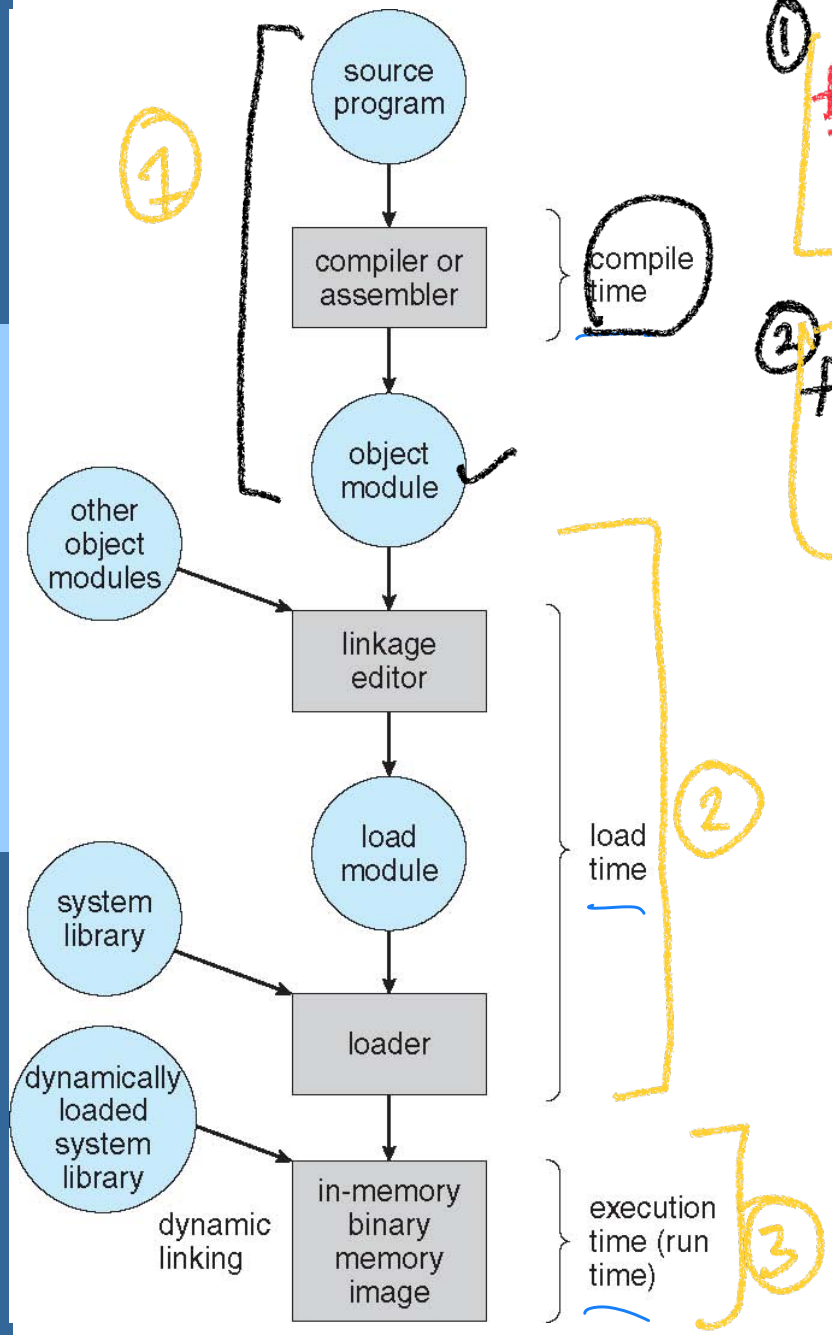
Address Binding (3)

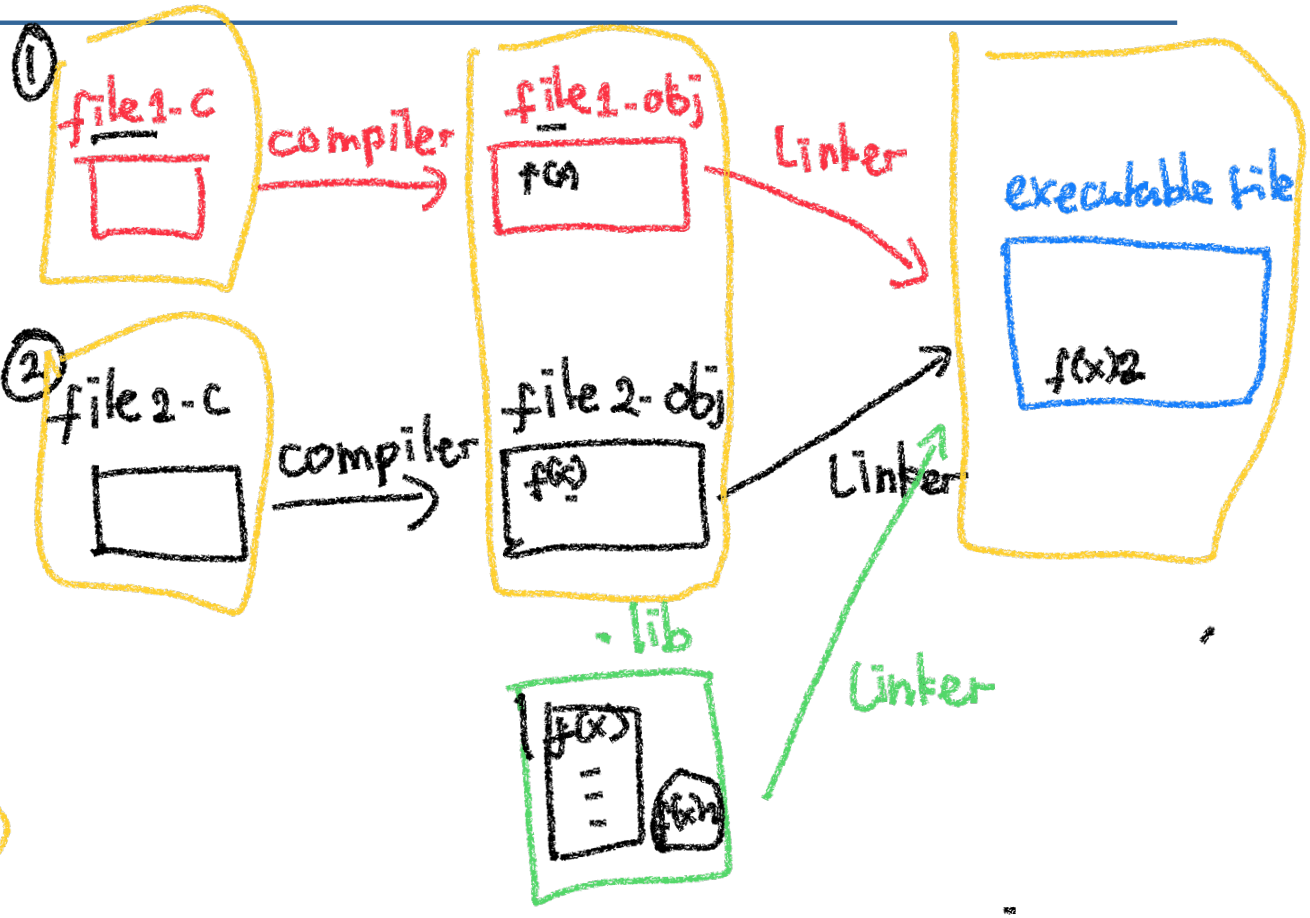


Binding of Instructions and Data to Memory

- Address binding of instructions and data to memory addresses can happen at three different stages
  - **Compile time**: If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
  - **Load time**: Must generate **relocatable code** if memory location is not known at compile time
  - **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another
    - Need hardware support for address maps (e.g., base and limit registers)

Multistep Processing of a User Program



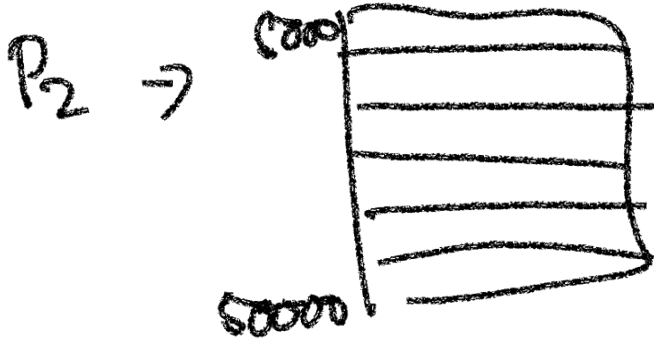


Static linking → happens before the program runs

Dynamic linking → happens during run time

Logical vs. Physical Address Space

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



Logical vs. Physical Address Space

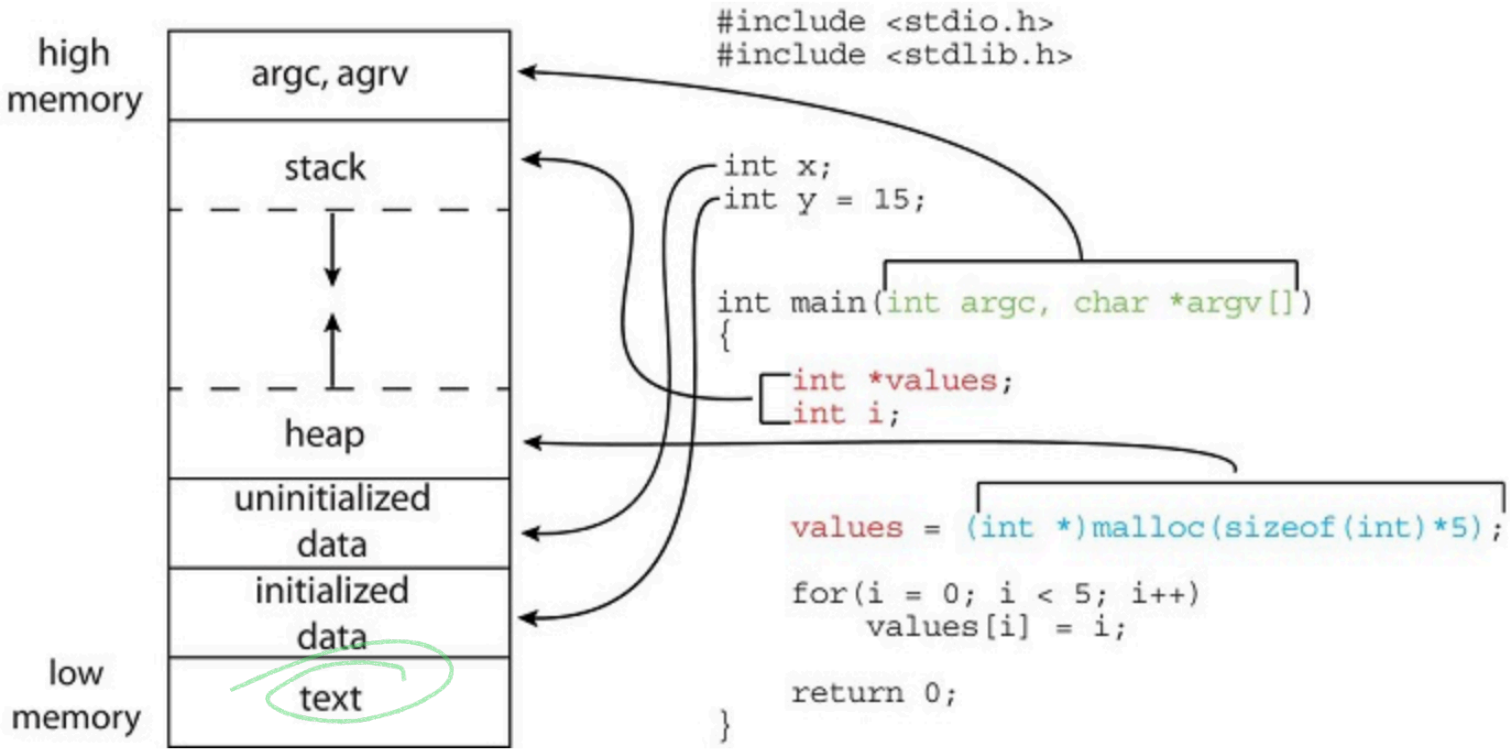
- The user program deals with *logical* addresses; it never sees the *real* physical addresses
  - Execution-time binding occurs when reference is made to location in memory
  - Logical address bound to physical addresses

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    printf("location of code : %p\n", main);
    printf("location of heap : %p\n", malloc(100e6));
    int x = 3;
    printf("location of stack: %p\n", &x);
    return 0;
}
```

Recall Memory Layout of a C Program

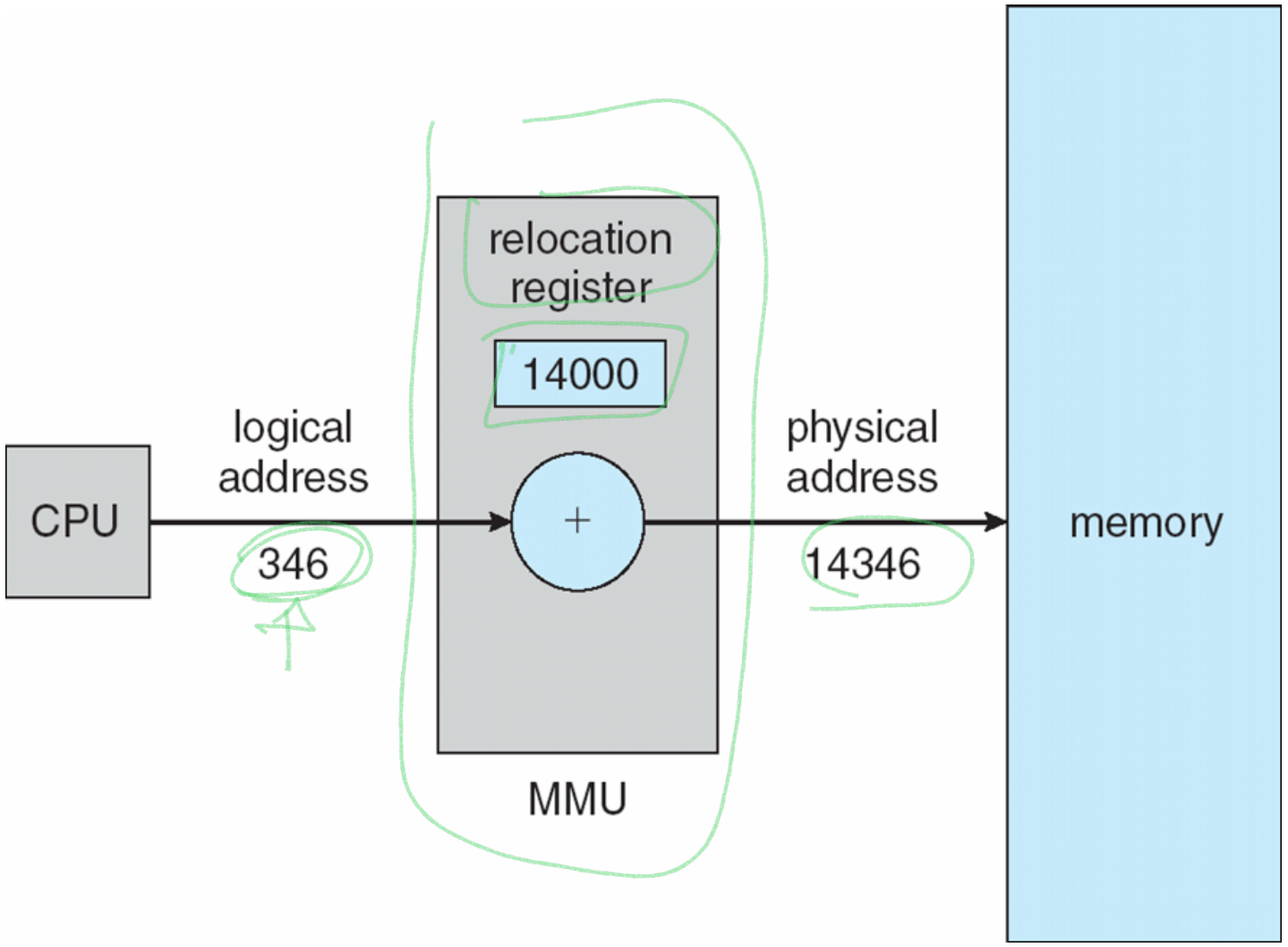
- The global data section is divided into different sections for (a) initialized data and (b) uninitialized data.
- A separate section is provided for the **`argc`** and **`argv`** parameters passed to the **`main()`** function.



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
- To start, consider 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
  - **Base register** now called **relocation register**
  - MS-DOS on Intel 80x86 used 4 relocation registers

MMU


relocation
register
14000
logical
address
346
CPU
+
physical
address
14346
memory
MMU

Dynamic relocation using a relocation register

- 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

-dll ~ Unix
.so ~ Windows
} for dynamic library

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

Swapping

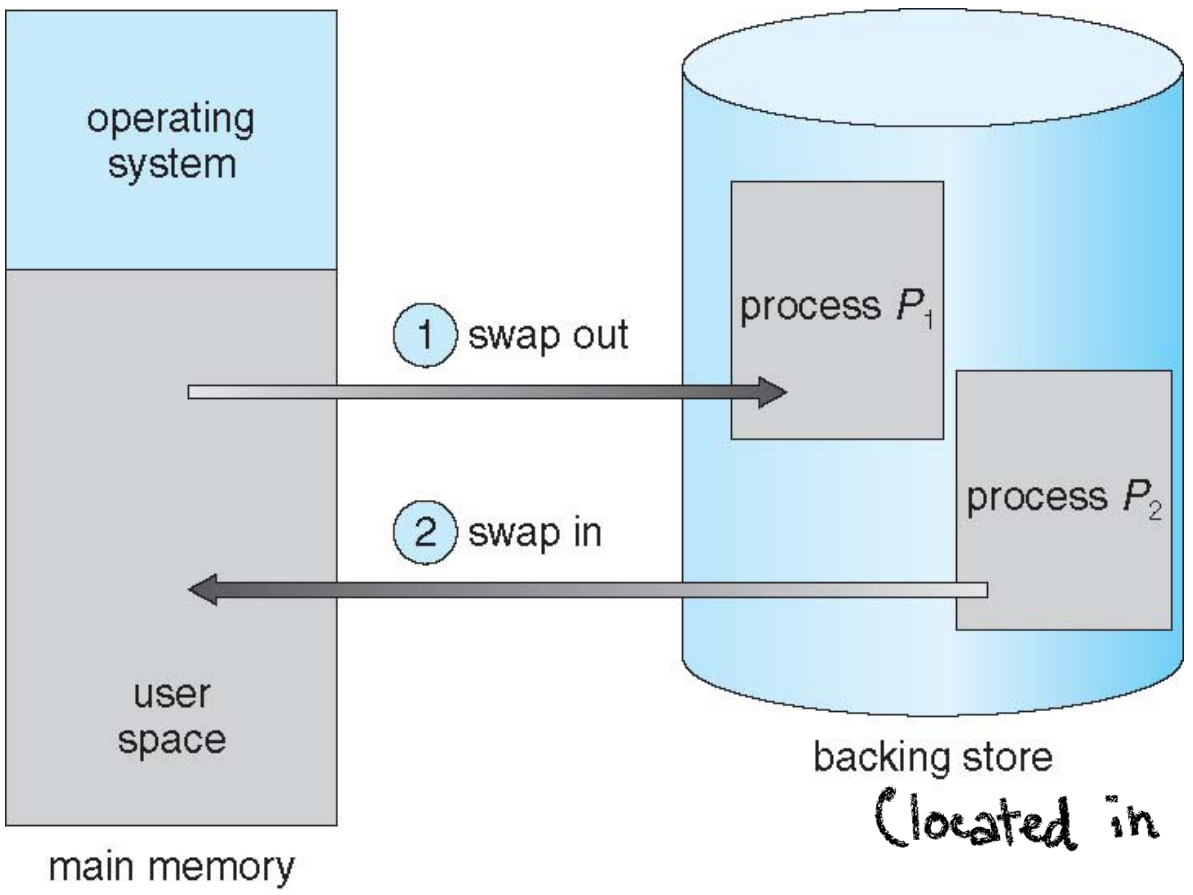
- A process can be **swapped** temporarily out of memory to a backing store, and then brought back into memory for continued execution
  - Total physical memory space of processes can exceed physical memory
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all 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

Schematic View of Swapping

Standard Swapping



(located in the disk)

Swapping is expensive, why? disk is slow

Context Switch Time including Swapping

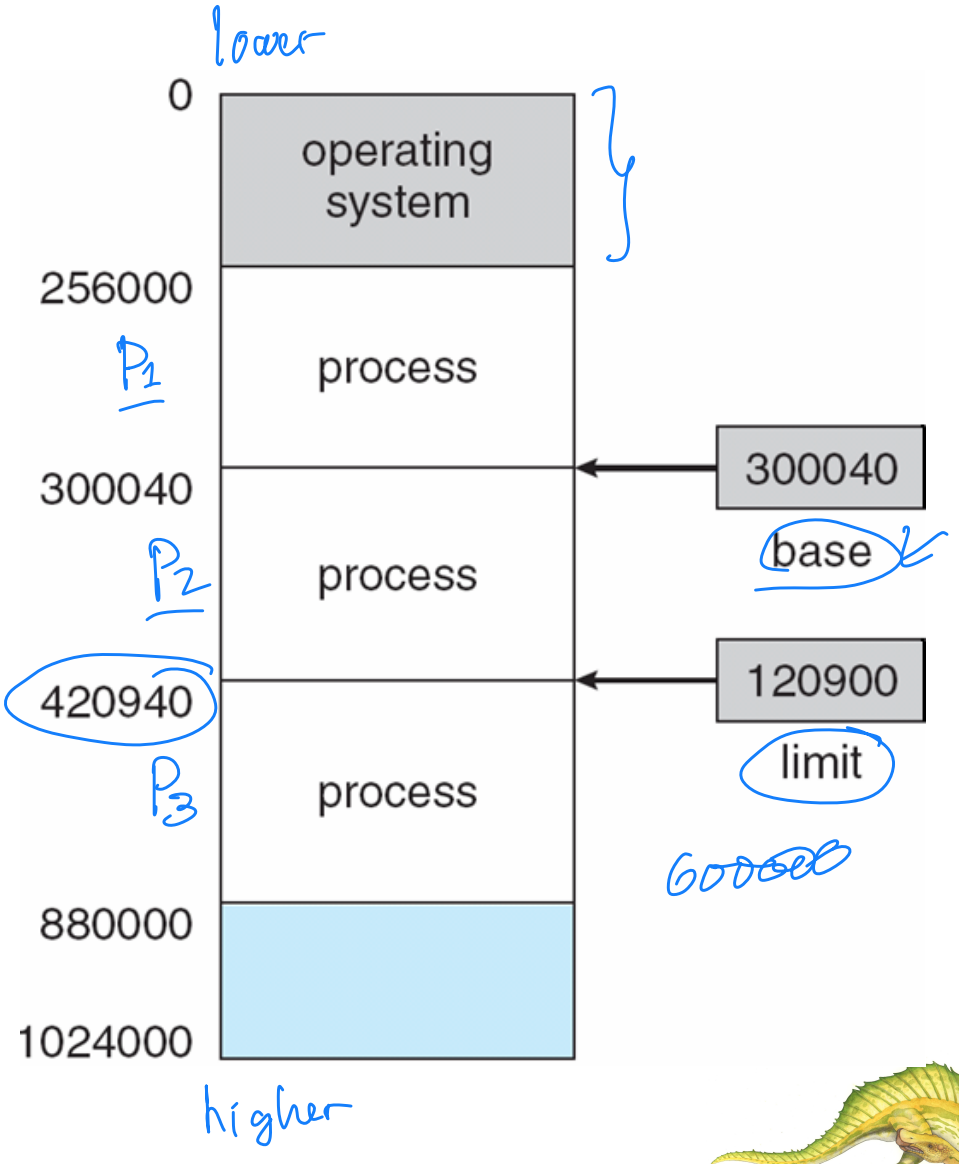
- If next processes to be put on CPU is not in memory, need to swap out a process and swap in target process
- Context switch time can then be very high
- 100MB process swapping to hard disk with transfer rate of 50MB/sec
  - Swap out time of 2000 ms
  - Plus swap in of same sized process
  - Total context 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**

Contiguous Allocation

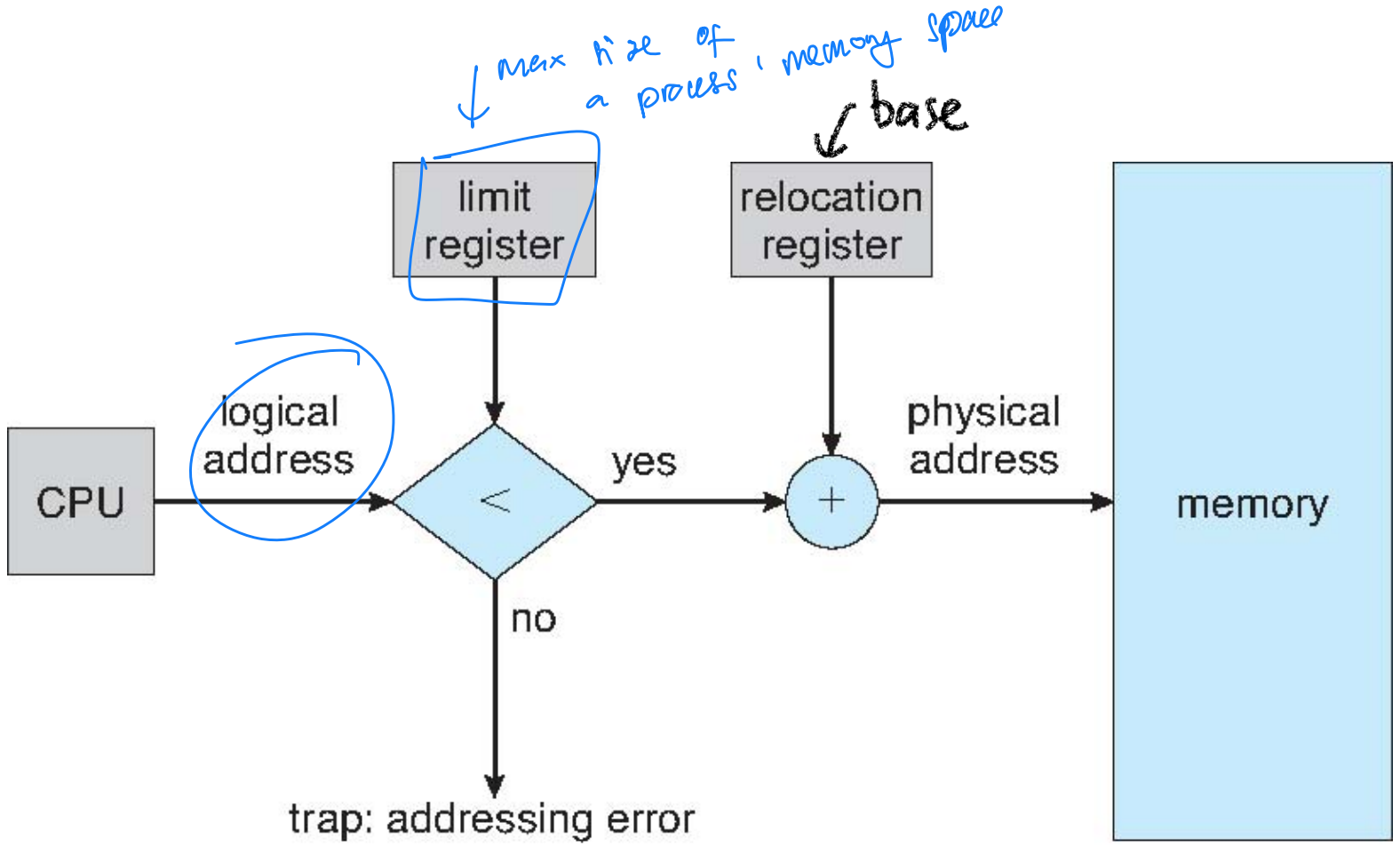
- Main memory must support both OS and user processes
- Limited resource, must allocate efficiently
- Contiguous allocation is one early method
- Main memory usually into two **partitions**:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory
  - Each process contained in single contiguous section of memory



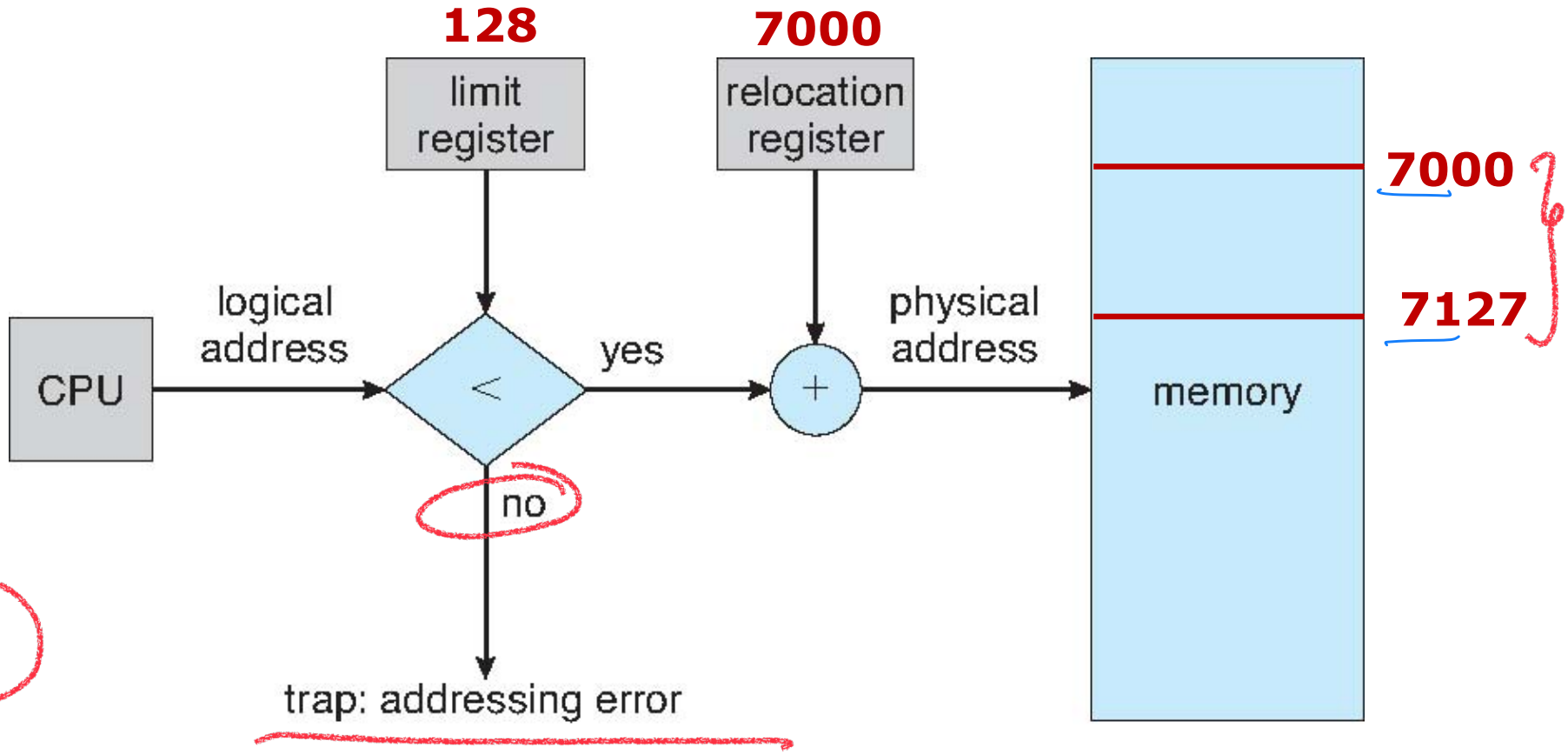
Contiguous Allocation (Cont.)

- 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*
  - Can then allow actions such as kernel code being **transient** and kernel changing size

Hardware Support for Relocation and Limit Registers



Hardware Support for Relocation and Limit Registers

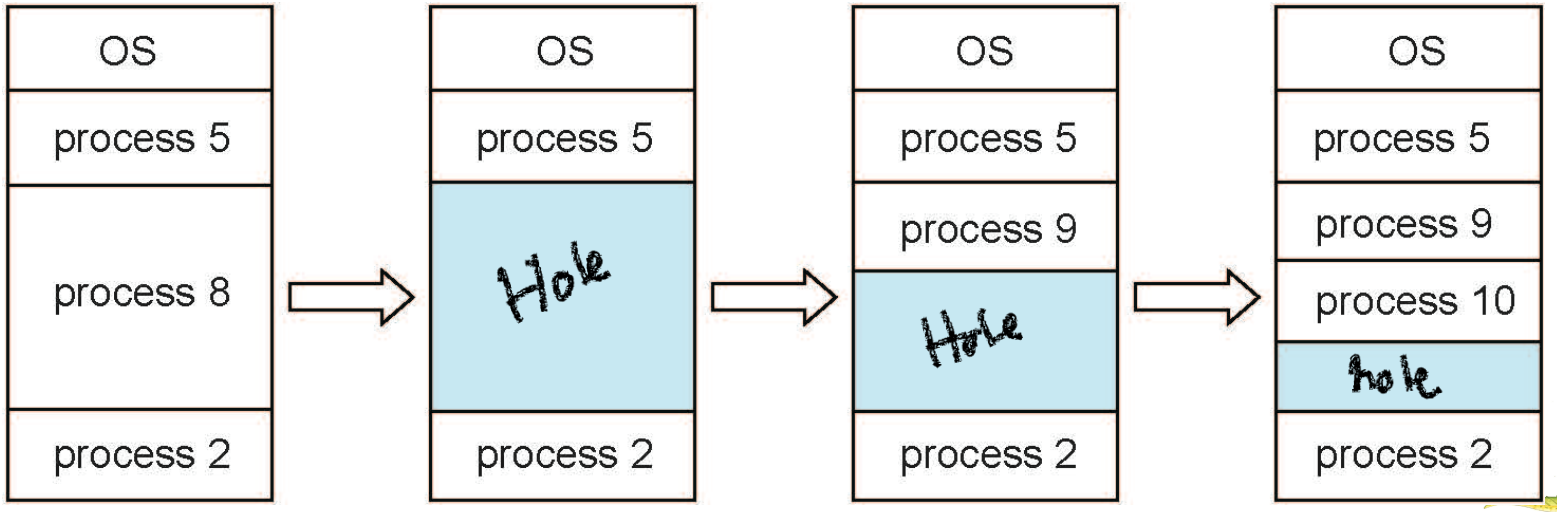


Suppose a process requires max 128 bytes and relocation register is set = 7000

In most modern systems, memory is **byte-addressable**, meaning each memory address points to a single byte (8 bits).

Multiple-partition allocation

- Multiple-partition allocation
  - Degree of multiprogramming limited by number of partitions
  - **Variable-partition** sizes for efficiency (sized to a given process' needs)
  - **Hole** – block of available memory; holes of various size are scattered throughout memory
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it
  - Process exiting frees its partition, adjacent free partitions combined
  - Operating system maintains information about:
    a) allocated partitions b) free partitions (hole)

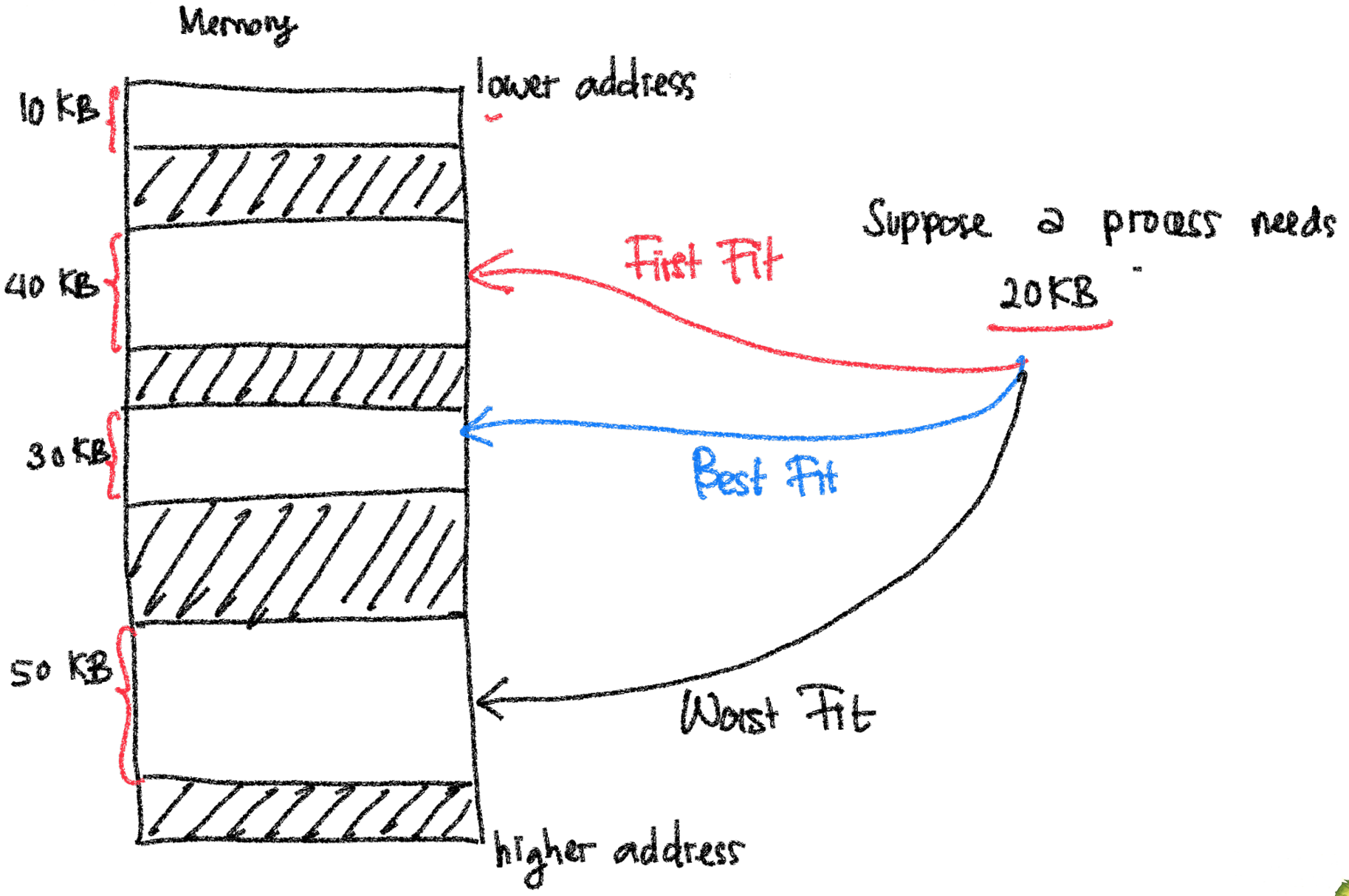


Dynamic Storage-Allocation Problem

How to satisfy a request of size ***n*** from a list of free holes?

- **First-fit**: Allocate the ***first*** hole that is big enough
- **Best-fit**: Allocate the ***smallest*** hole that is big enough; must search entire list, unless ordered by size
  - Produces the smallest leftover hole
- **Worst-fit**: Allocate the ***largest*** hole; must also search entire list
  - Produces the largest leftover hole

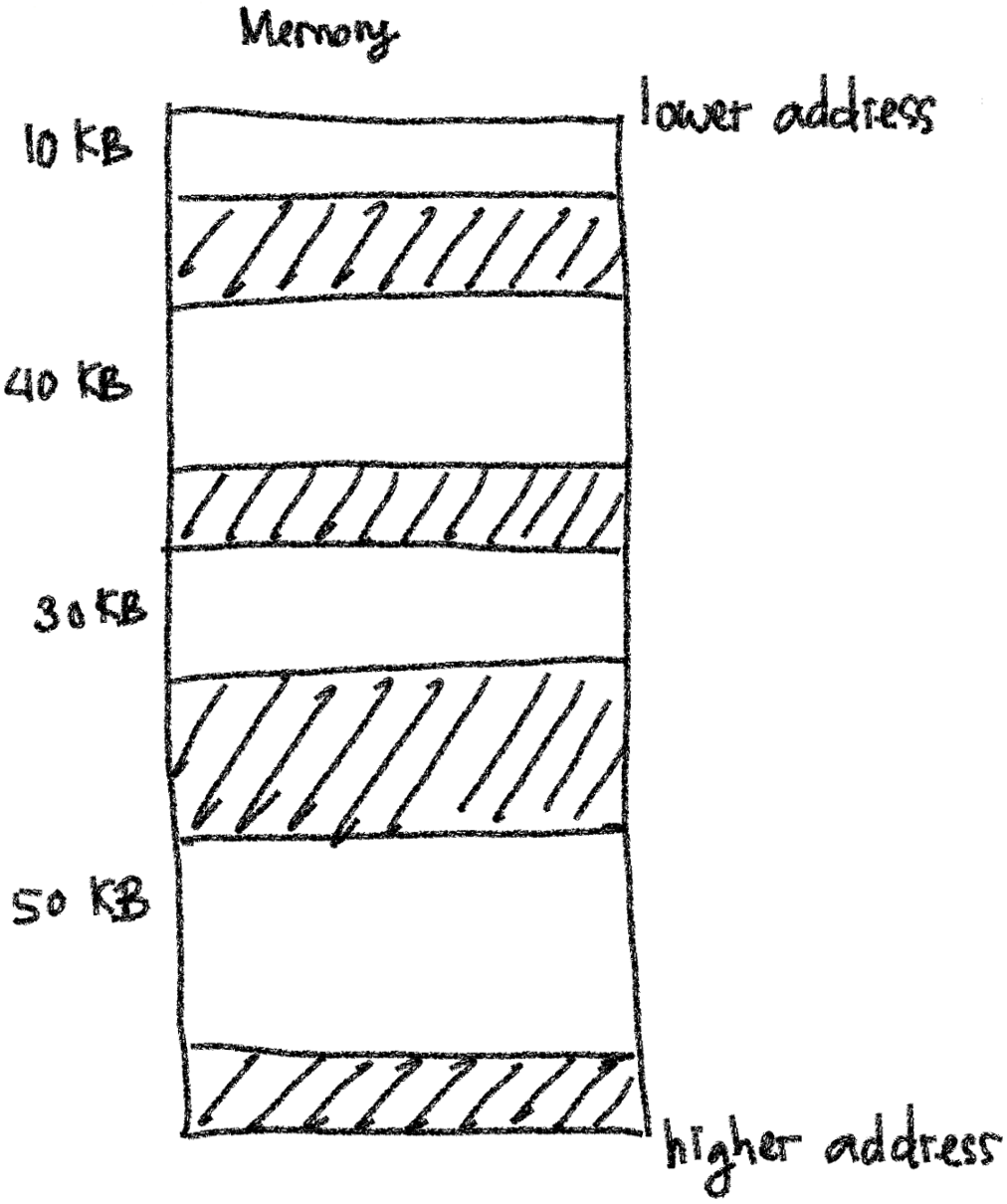
First-fit and best-fit better than worst-fit in terms of speed and storage utilization


Memory
lower address
10 KB
40 KB
30 KB
50 KB
higher address
Suppose a process needs
20 KB
First Fit
Best Fit
Worst Fit

Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- First fit analysis reveals that given *N* blocks allocated, 0.5 *N* blocks lost to fragmentation
  - 1/3 may be unusable -> **50-percent rule**

External Fragmentation

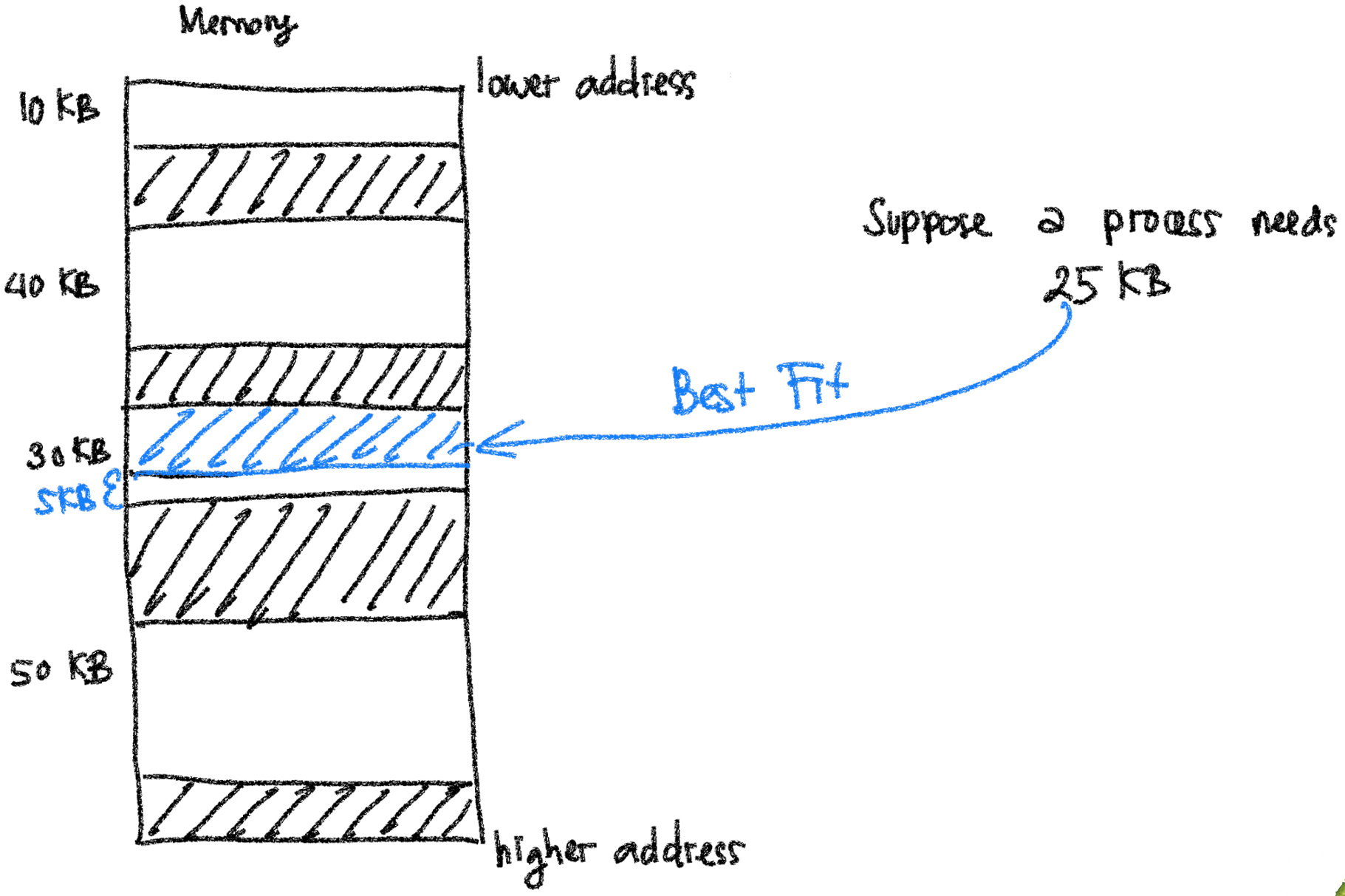


Suppose a process needs 130 KB

Fragmentation

- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- Consider a multiple-partition allocation scheme with a hole of 18,464 bytes. Suppose that the next process requests 18,462 bytes. If we allocate exactly the requested block, we are left with a hole of 2 bytes. The overhead to keep track of this hole will be substantially larger than the hole itself.
- The general approach to avoiding this problem is to break the physical memory into fixed-sized blocks and allocate memory in units based on block size.

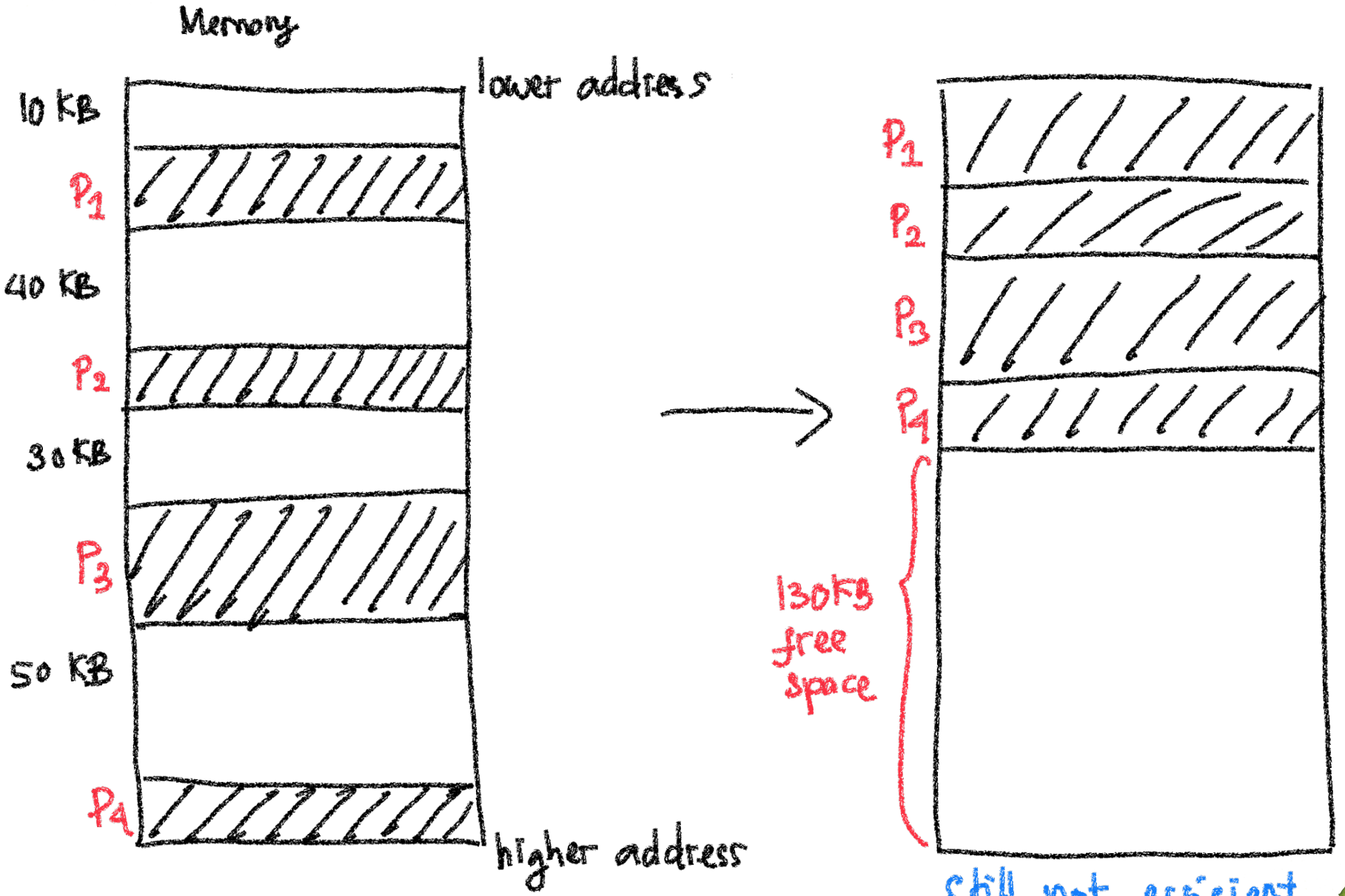
Internal Fragmentation

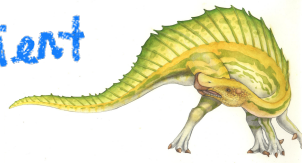


Fragmentation (Cont.)

- 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
- Now consider that backing store has same fragmentation problems

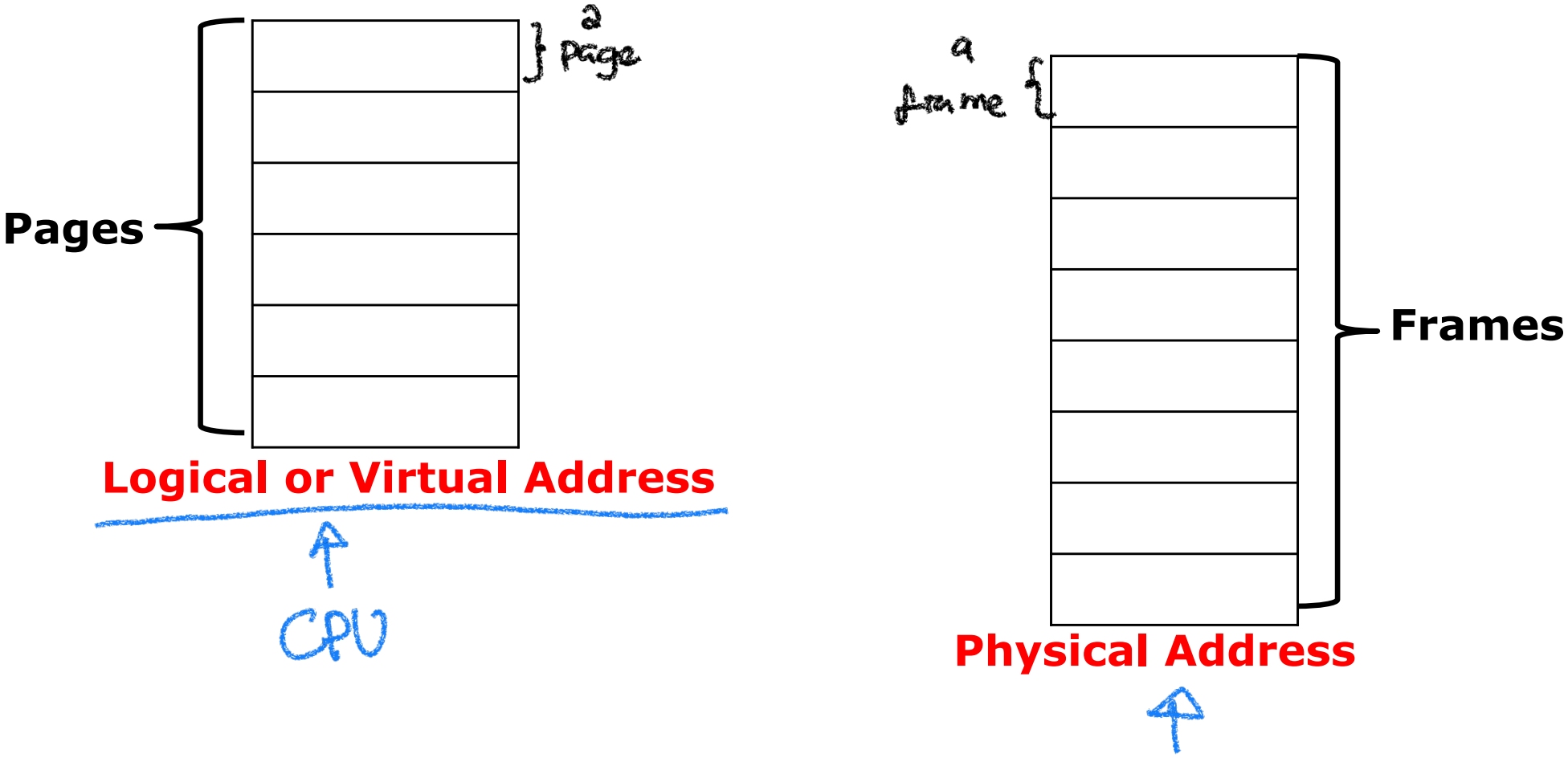
Compaction


Memory
lower address
10 KB
P1
40 KB
P2
30 KB
P3
50 KB
P4
higher address
P1
P2
P3
P4
130KB
free
space
still not efficient
why?



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** (per process) to translate logical to physical addresses
- Backing store (**used for Swapping**) likewise split into pages
- Still have Internal fragmentation



page size = frame size

a page ⟶ a frame

Aside:

1 KB

A frame size is power of 2, between **512 bytes** (**2^9 bytes**) and **16 Mbytes** ($2^4 * 2^{20} = 2^{4+20} =$ **2^{24} bytes**)

Computers use binary addressing, meaning that each memory address corresponds to a unique binary number.

e.g. 6 bits space:

1	0	0	1	0	0

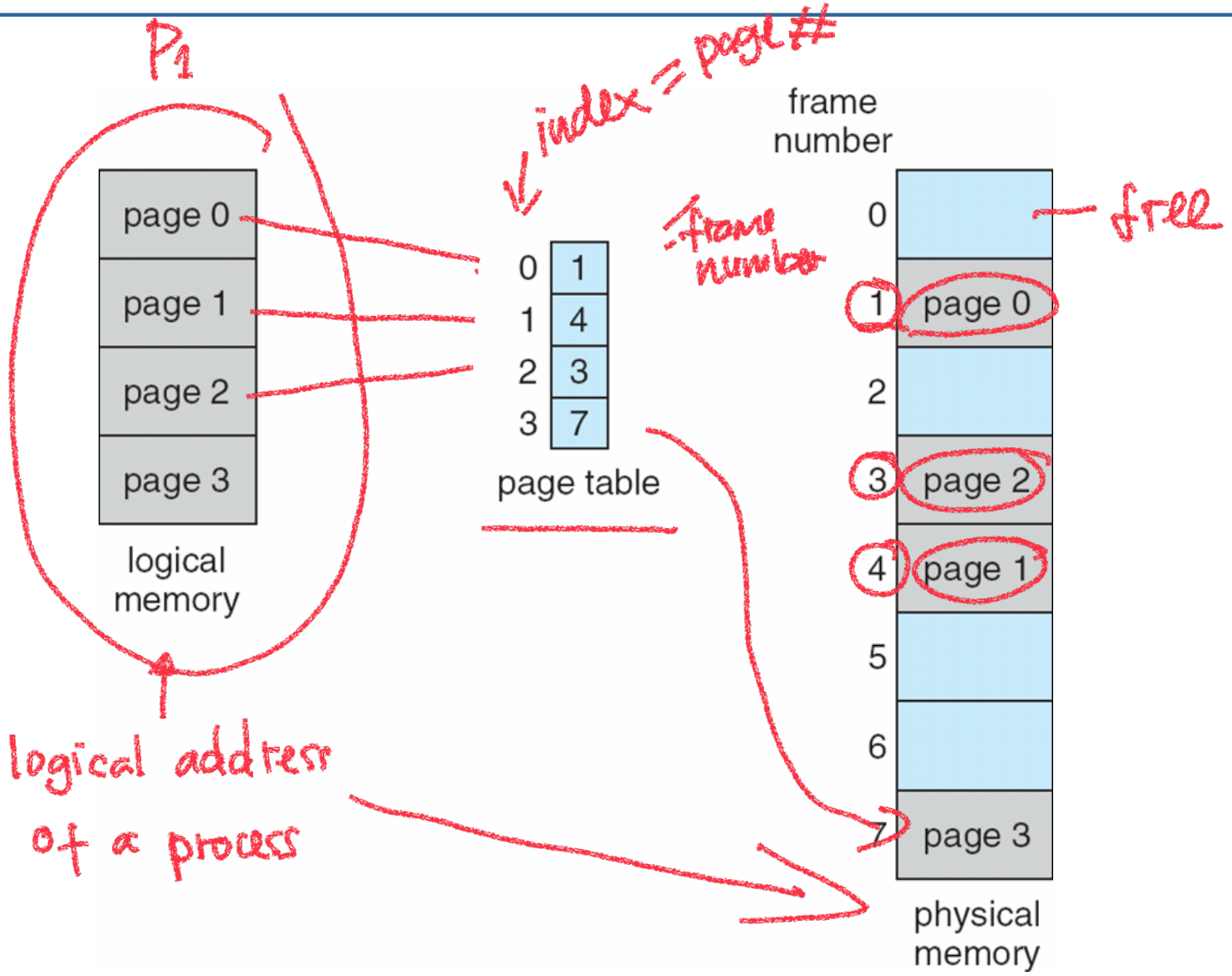
2^6 Bytes Possible addresses

1 KB = 2^{10} Bytes

1 MB = 2^{20} Bytes

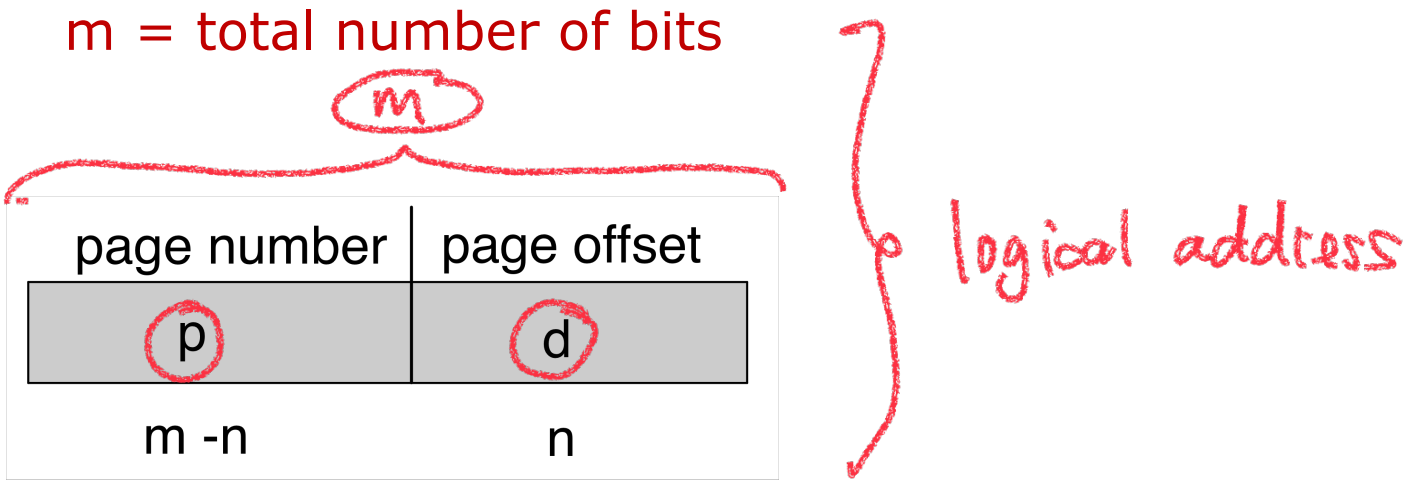
1 GB = 2^{30} Bytes

Paging Model of Logical and Physical Memory



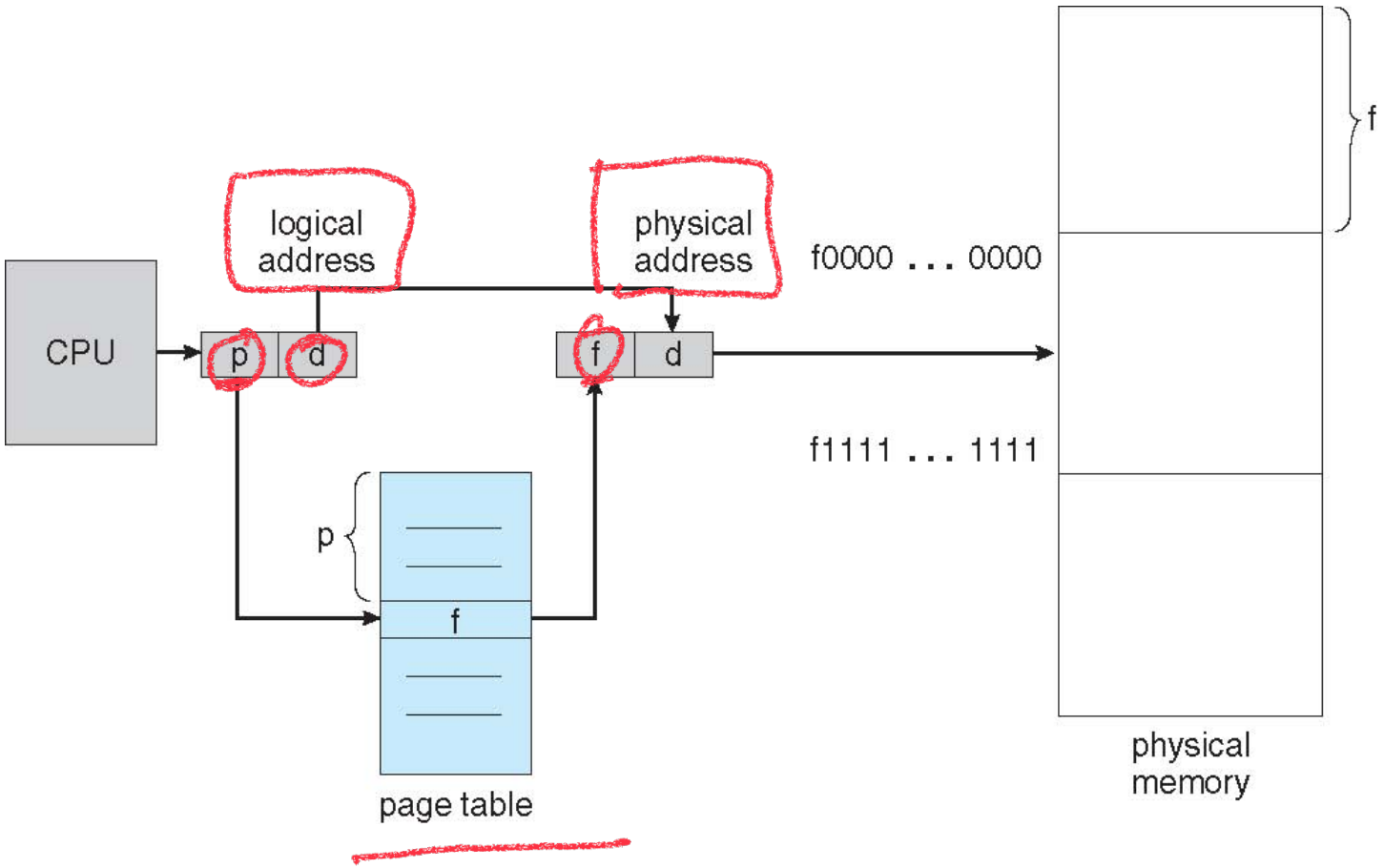
Address Translation Scheme

- Address generated by CPU (**logical address**) 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



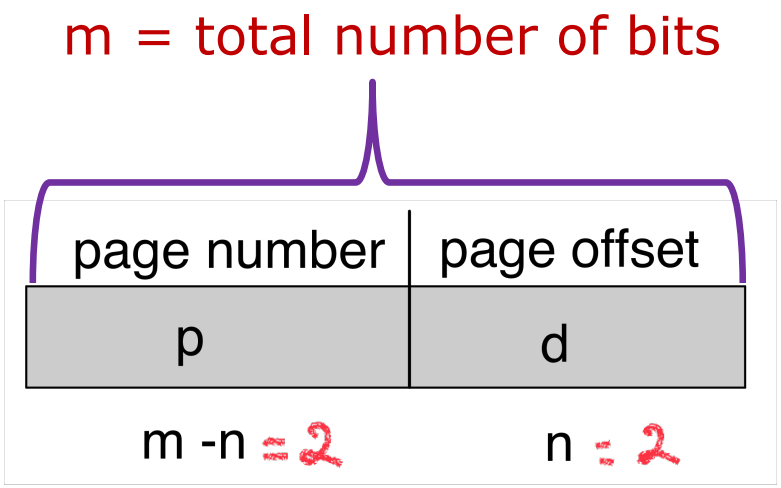
  - For given logical address space 2^m
    - page size 2^n → page offset → determine the size of the page from
    - page numbers 2^{m-n} → determine # of entries in the page table

Paging Hardware



Example:

m = total number of bits = 4



Suppose:
m = 4, n = 2

Then:
the total number of pages = $2^{m-n} = 2^{4-2} = 4$ 2^2
page size = $2^n = 2^2 = 4$ Bytes

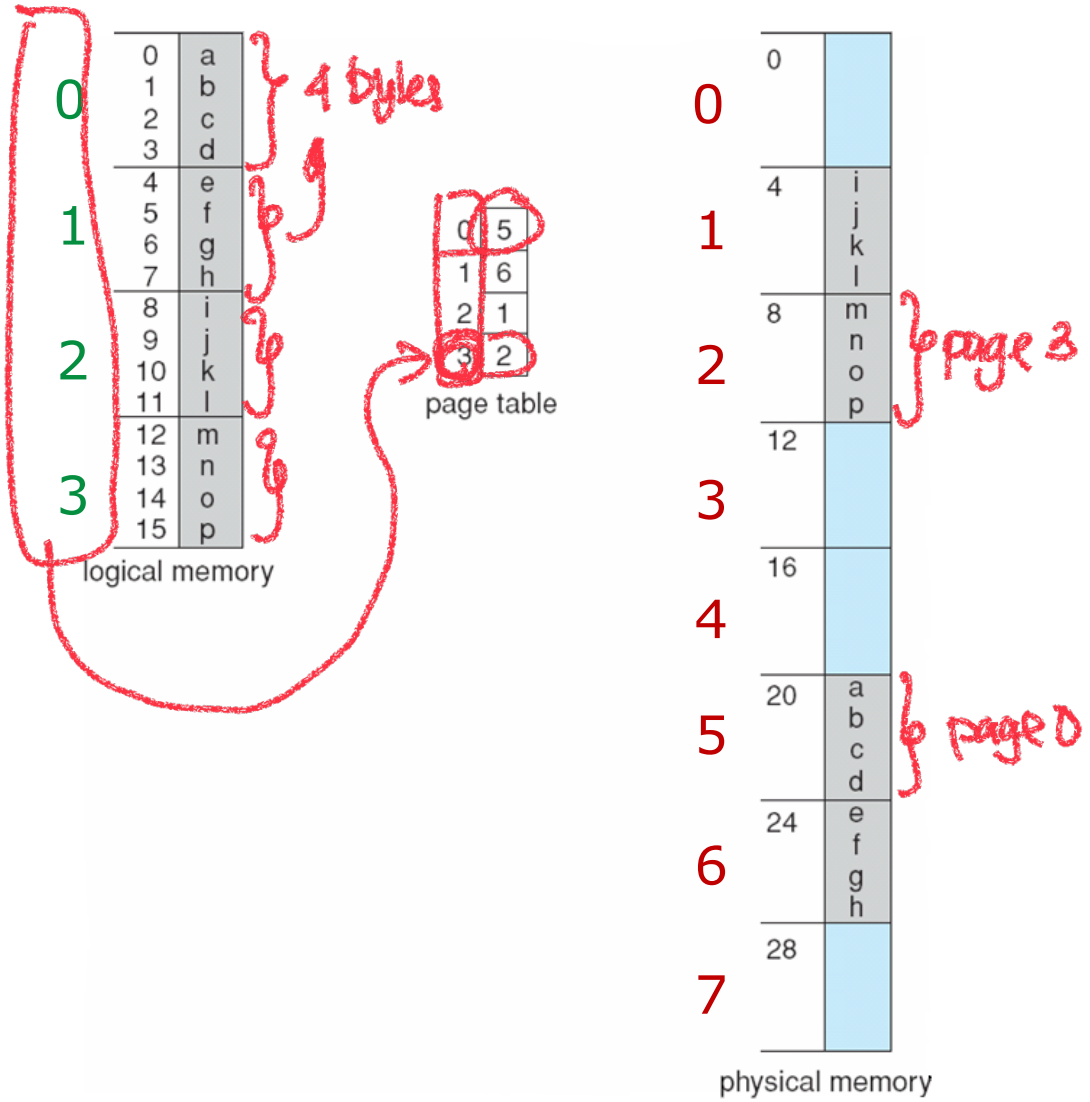
Paging Example

page number	page offset
p	d
m -n	n

n=2 and *m*=4

Total number of pages = $2^{m-n} = 2^2 = 4$

Each page contains $2^n = 2^2 = 4$ bytes (page size)



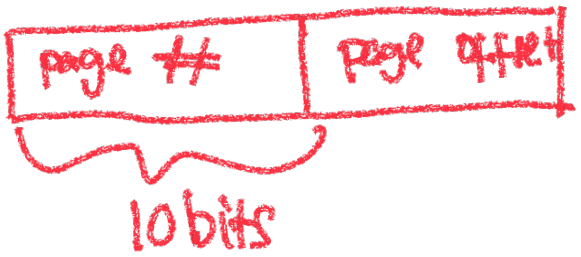
32 bytes (8 frames)

Check your understanding

1. **A page number** is represented by 10 bits. How many entries are in the page table?

   A) 10

   B) 2^{10}

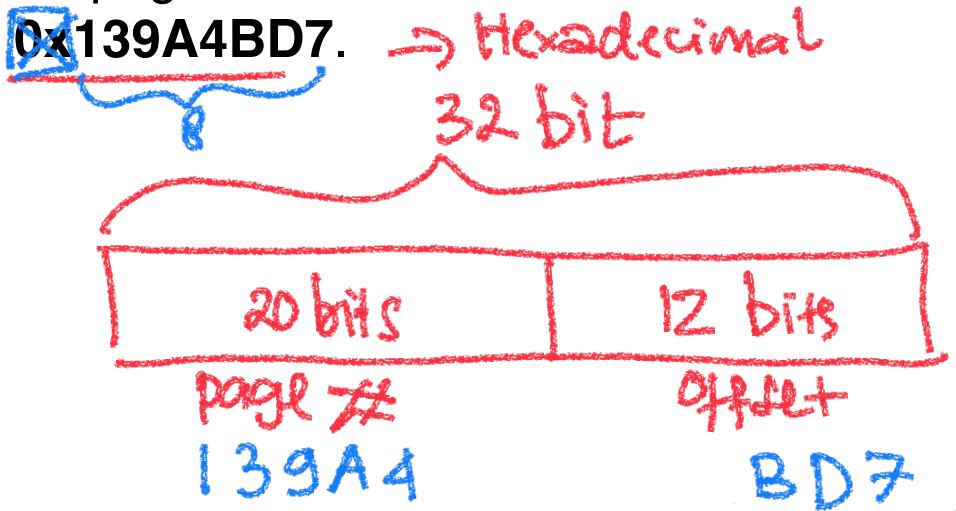


2. A 32-bit address is divided into 20 bits for the page number and 12 bits for the offset. _____ is the page number and _____ is the offset for the logical address **0x139A4BD7**.

   A) **0x139, 0xA4BD7**

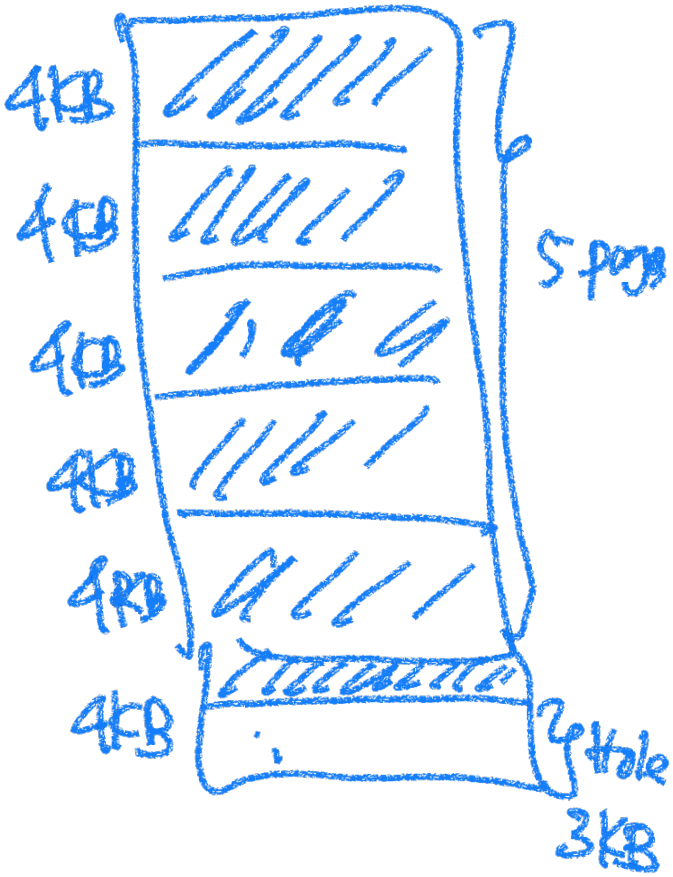
   B) **0x139A, 0x4BD7**

   C) **0x139A4, 0xBD7**



21 KB

21/4 = 5.25



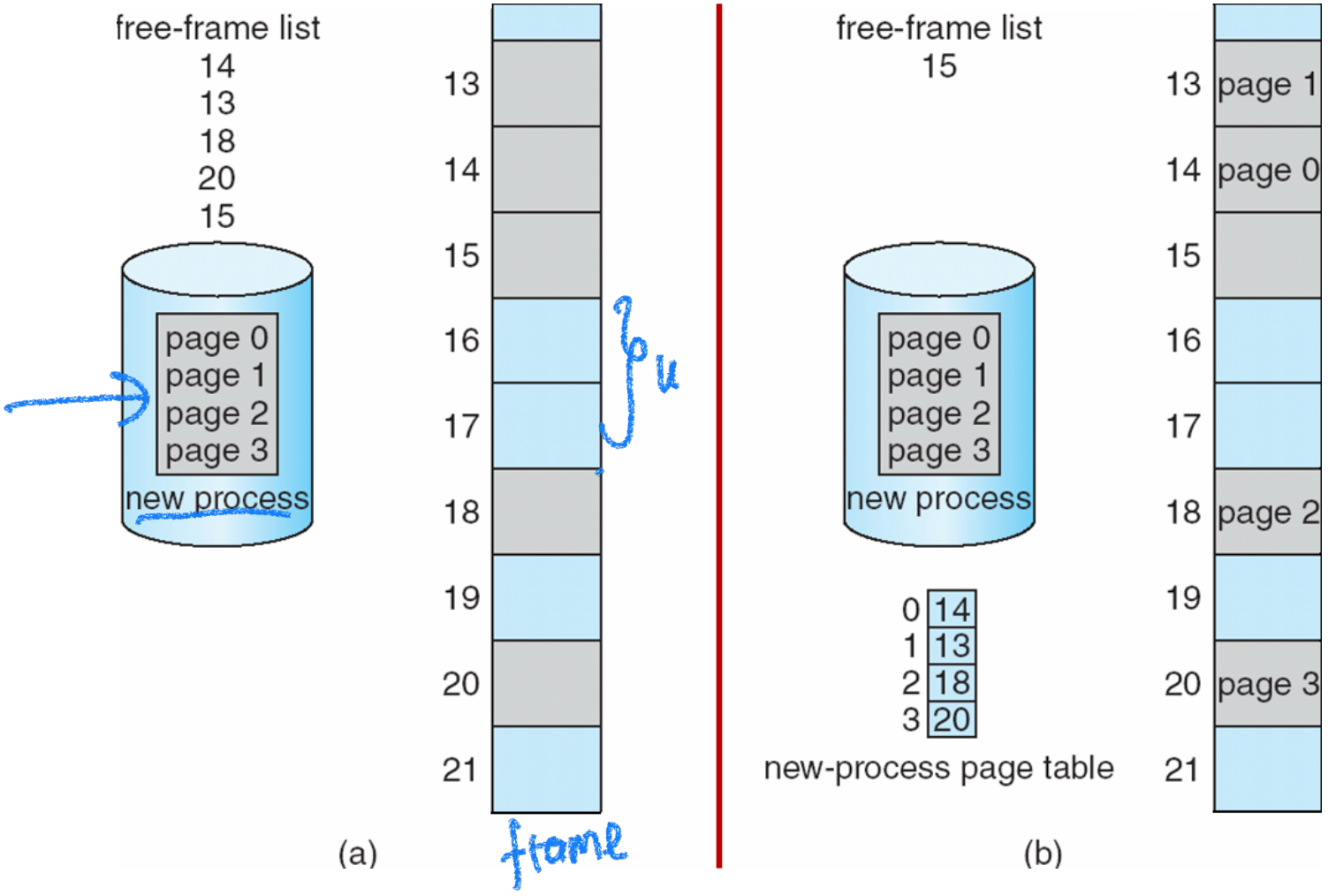
Paging (Cont.)

- Calculating internal fragmentation
  - Page size = 2,048 bytes = 2KB
  - Process size = 72,766 bytes
  - #pages requires by the process = 36 pages
    - **72,766/ 2,048 = 35 pages + 1,086 bytes = 36 pages**
    - 35 * 2,048 Bytes + 1,086 Bytes = 72,766 Bytes
  - Internal fragmentation of 2,048 - 1,086 = 962 bytes → page 36
  - 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 (typically 4 KB or 8 KB in size)
    - Solaris supports two page sizes – 8 KB and 4 MB
    - Windows 10 supports page sizes of 4 KB and 2 MB

Paging (Cont.)

- Process view and physical memory now very different
  - An important aspect of paging is the clear separation between the programmer's view of memory and the actual physical memory.
  - In fact, the user program is scattered throughout physical memory, which also holds other programs.
- By implementation process can only access its own memory

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
- **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**)

Implementation of Page Table (Cont.)

- 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
    - Used LRU (least recently used), round robin, or random
  - Some entries can be **wired down** for permanent fast access

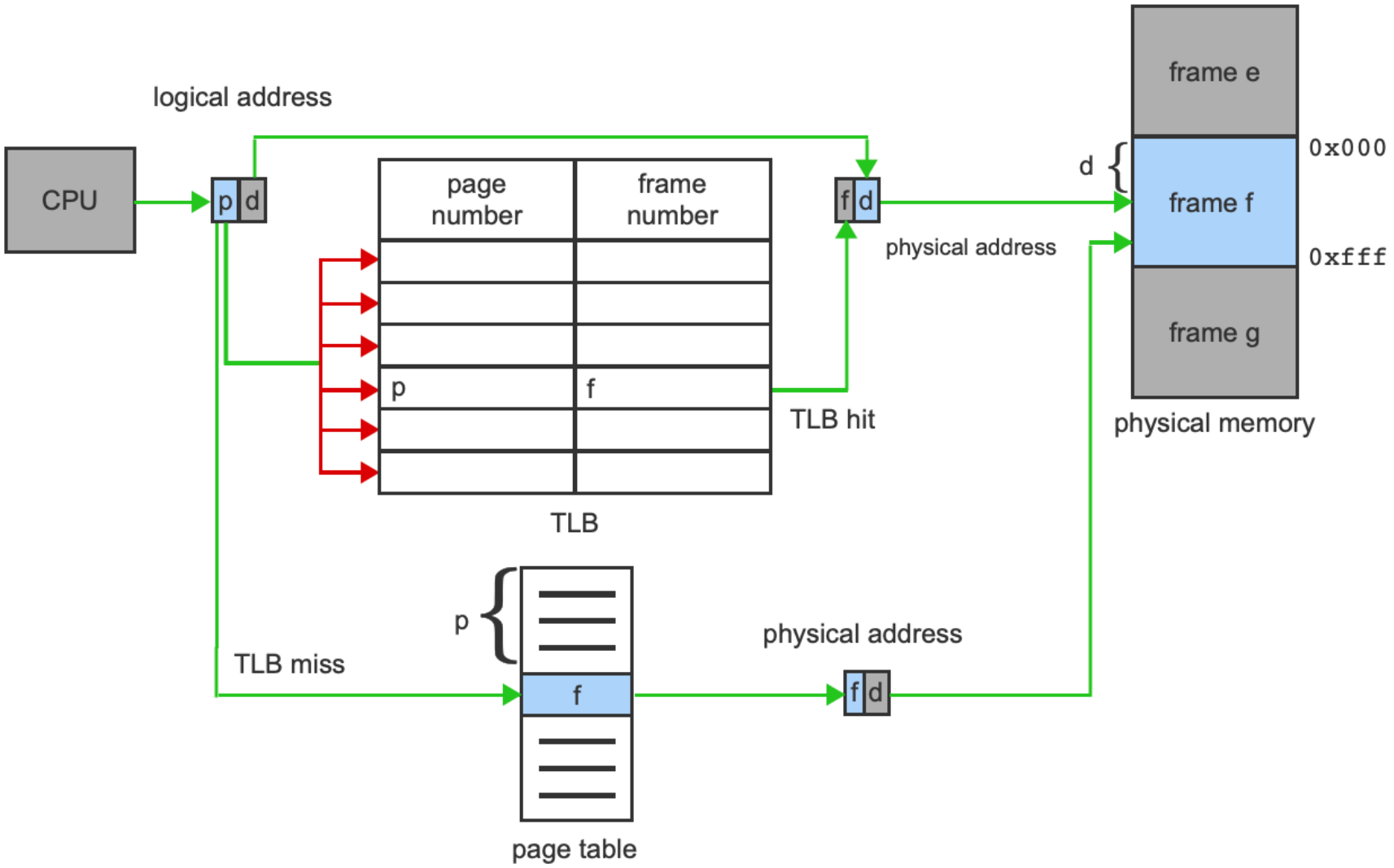
Associative Memory (TLBs)

- Associative memory – parallel search

Page #	Frame #

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

Paging Hardware With TLB



Effective Access Time

- 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 register
- **Effective Access Time** (**EAT**)

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

  (when ε is omitted)

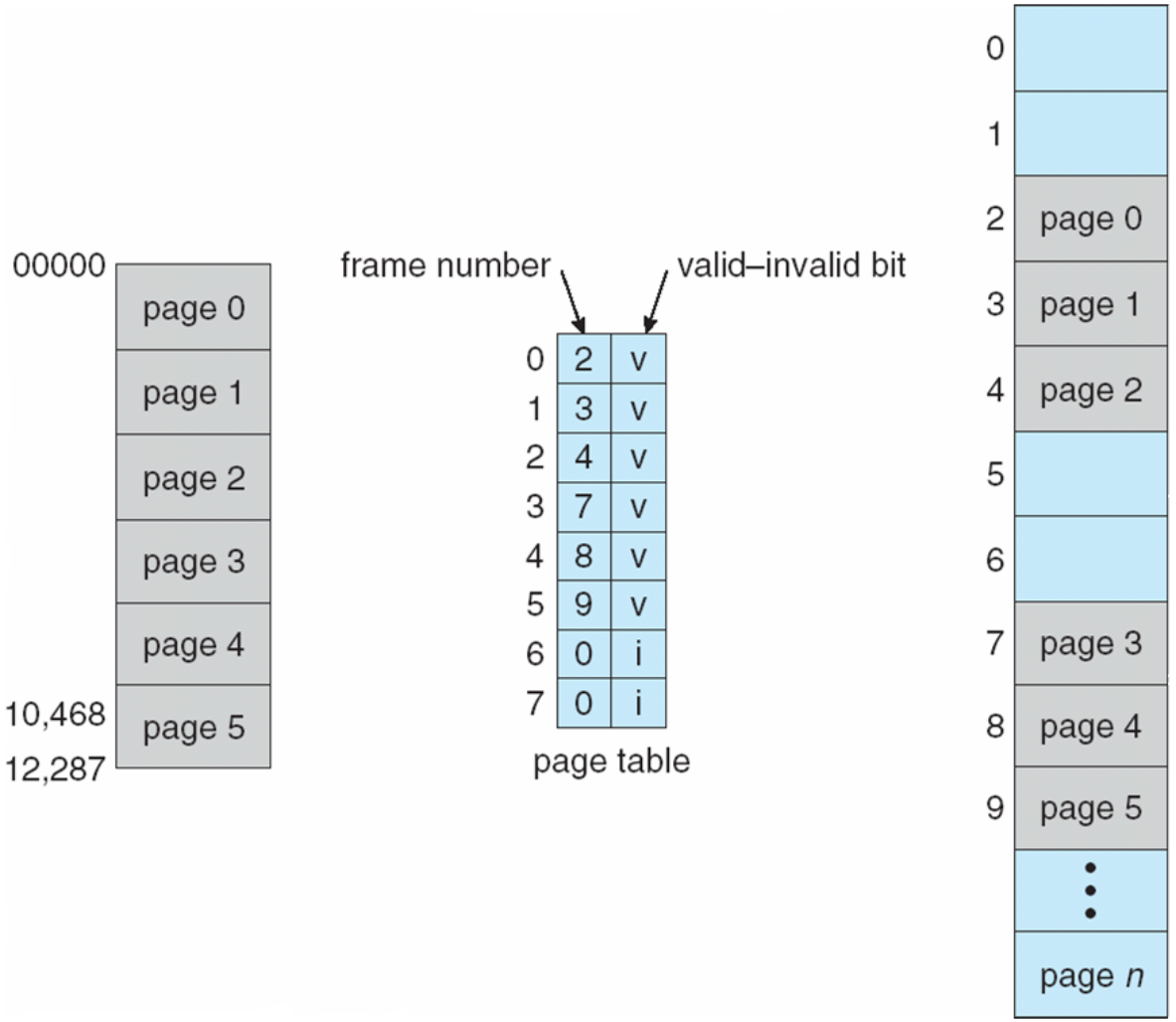
EAT = α x (1 x memory access time) + (1- α) x (2 x memory access time)

- Consider α = 80%, ε = 20ns for TLB search, 100ns for memory access
  - EAT = **0.80 x 100** + **0.20 x 200** = 120ns (ε is omitted)
- Consider more realistic hit ratio -> α = 99%, ε = 20ns for TLB search, 100ns for memory access
  - EAT = 0.99 x 100 + 0.01 x 200 = 101ns

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

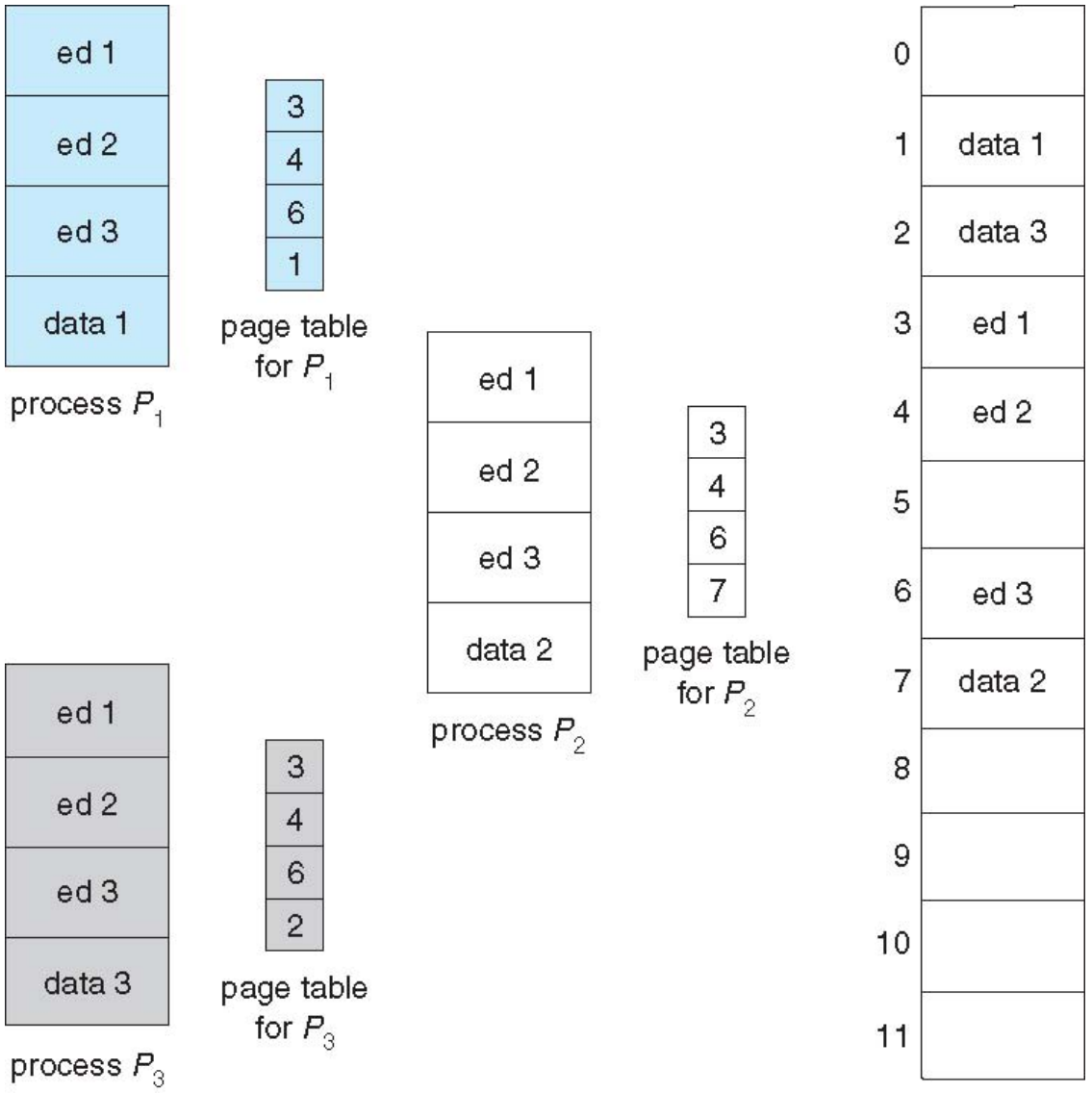
Valid (v) or Invalid (i) Bit In A Page Table



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



Check your understanding

1) A logical address space of size 2^32 with a page size of 2^12 requires _____ bits for the page number and _____ bits for the page offset.

A) 12, 20

B) 20, 12

C) 32, 12

2) How many bits must be used to represent the page offset for a page size of 8 KB?

A) 8

B) 10

C) 13

3) Paging can have _____ fragmentation but not _____ fragmentation.

A) internal, external

B) external, internal

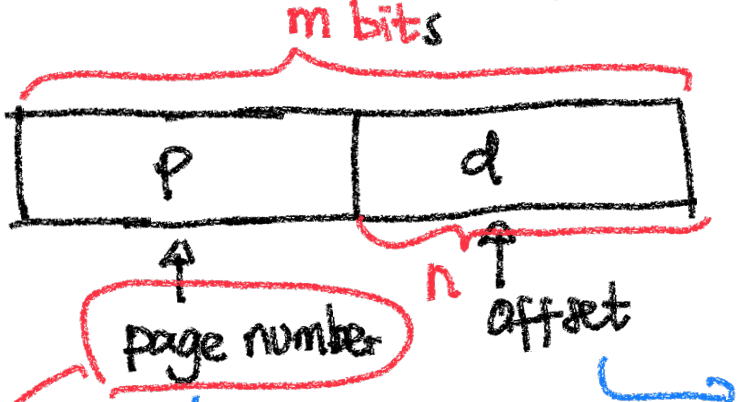
Check your understanding

6) What is the effective memory access time with a TLB hit ratio of 85% and 15 ns memory access time?

7) What is the page offset of the logical address 0xAEF9 with a page size of 256 bytes?

Paging

Recall the structure of logical address in paging



→ determine the size of each page or frame

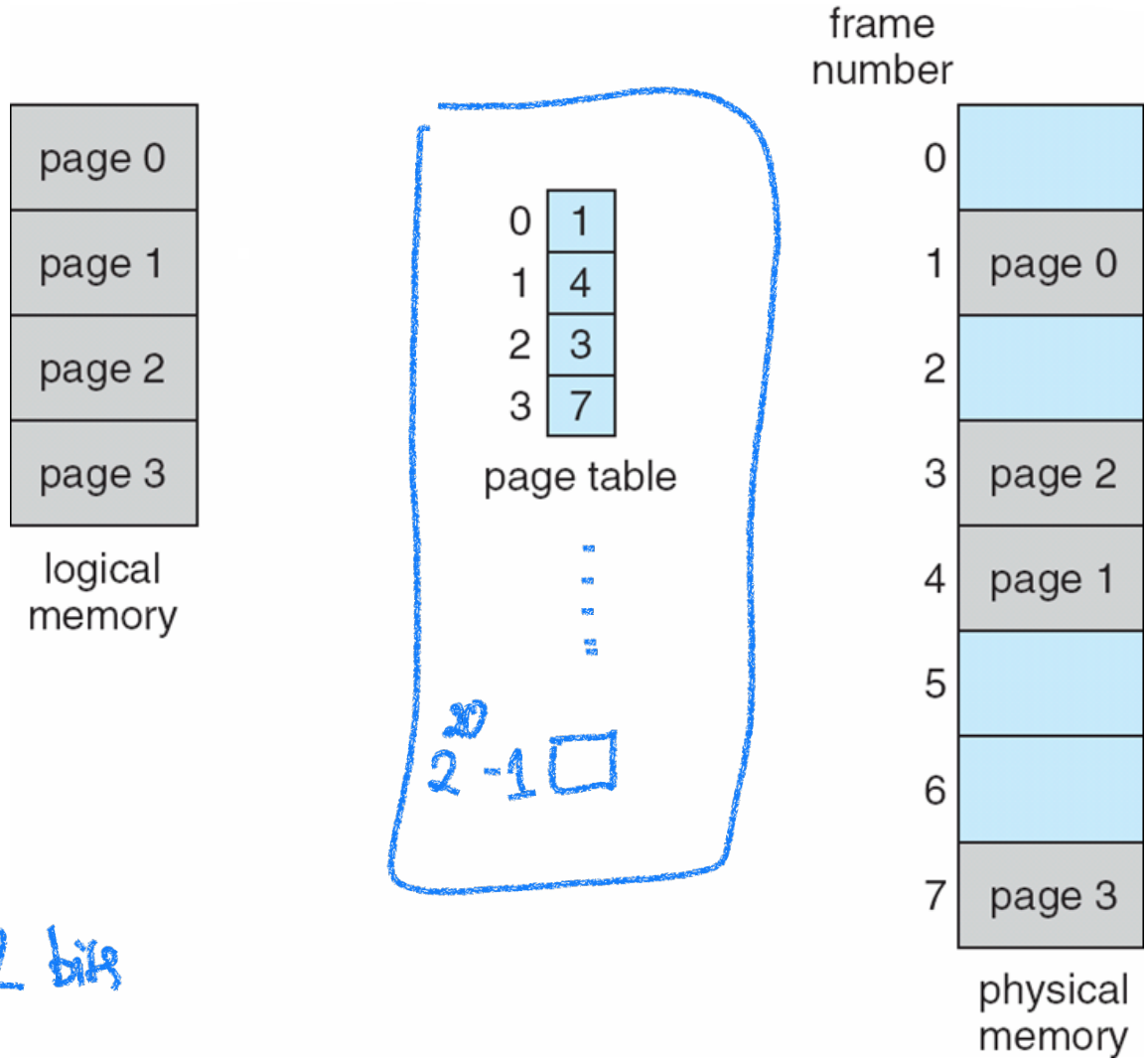
→ determine the number of entries in the page table

If offset d is n bits → the size of each page = ? 2^n

1 address ≃ 1 byte

m-n bits → # entries in page table = 2^{m-n}

Paging Model of Logical and Physical Memory

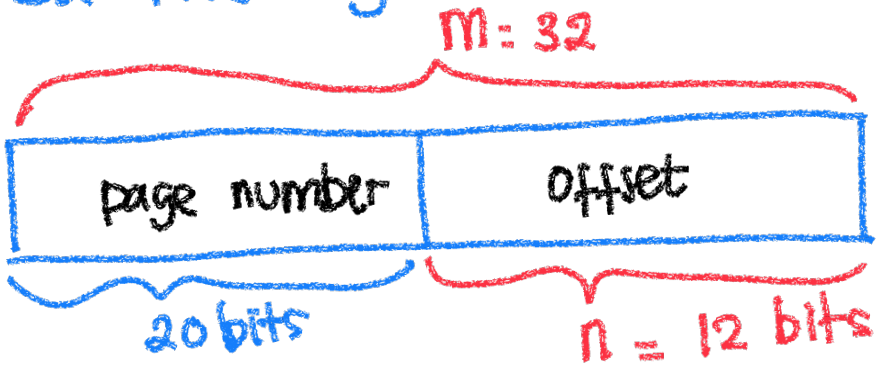


$1\,KB = 2^{10}\,B$

$1\,MB = 2^{20}\,B$

$1\,GB = 2^{30}\,B$

32 bits logical address



Page size = 4KB ≈ offset

$n \Rightarrow 4KB = 2^2 \times 2^{10}$ Bytes $= 2^{12}$ Bytes

$n = 12$

How many bits used for page number? $32 - 12 = 20$

What is the size of the page table? 2^{20} Bytes $= 1\,MB$

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}) → 4×2^{10} Bytes = $2^2 \times 2^{10}$ Bytes
  - Page table would have 1 million entries ($2^{32} / 2^{12}$)
  - 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
    - Don't want to allocate that contiguously in main memory
- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

1 KB = 2^{10} Bytes

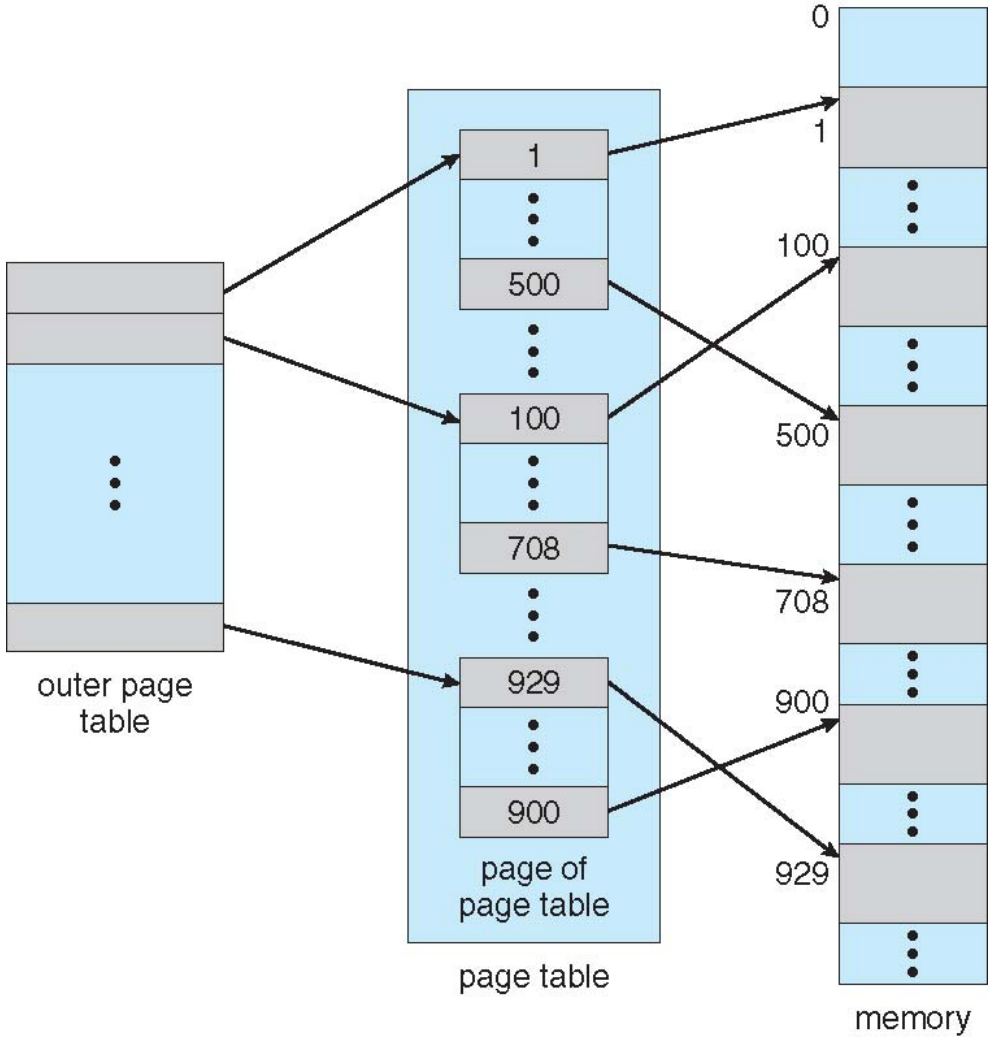
1 MB = 2^{20} Bytes

1 GB = 2^{30} Bytes

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



address translation works from the outer page table inward, also known as a ***forward-mapped*** page table.

Two-Level Paging Example

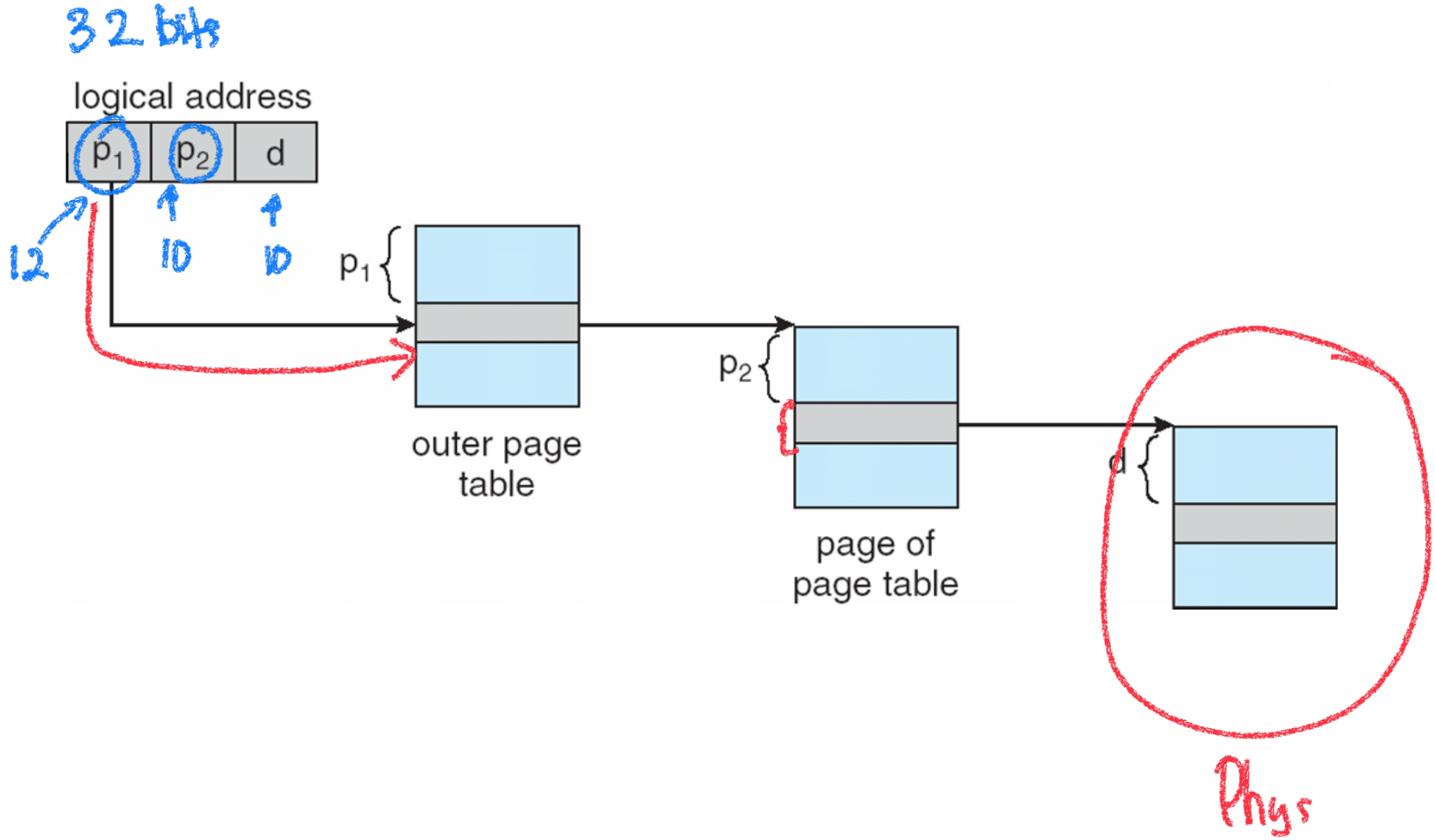
- A logical address (on 32-bit machine with 1K page size) is divided into:
  - a page number consisting of 22 bits
  - a page offset consisting of 10 bits
- Since the page table is paged, the page number is further divided into:
  - a 12-bit page number
  - a 10-bit page offset
- Thus, a logical address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

the size of each page

page number		page offset
p_1	p_2	d
12	10	10

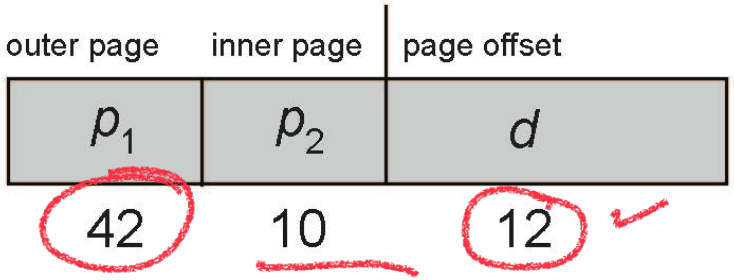
- where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table
- Known as **forward-mapped page table**

Address-Translation Scheme



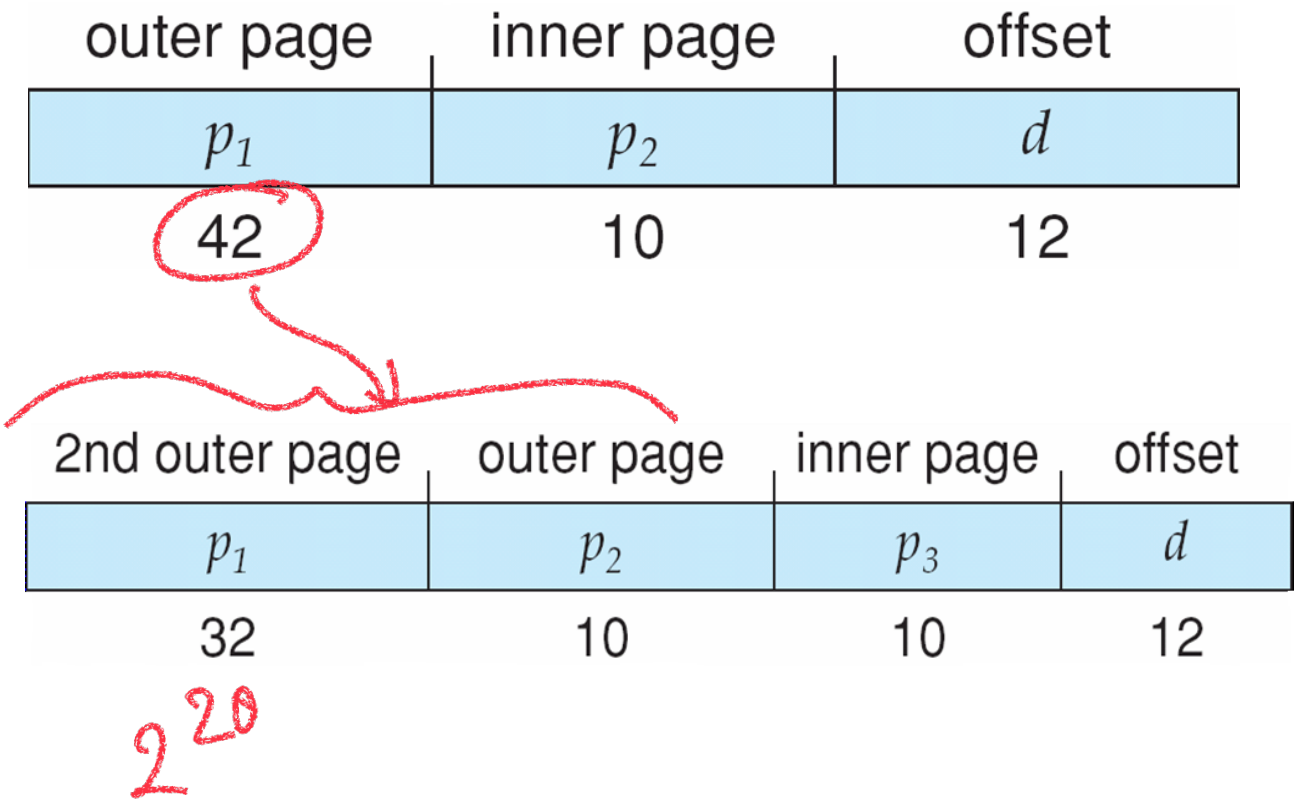
64-bit Logical Address Space

- 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 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

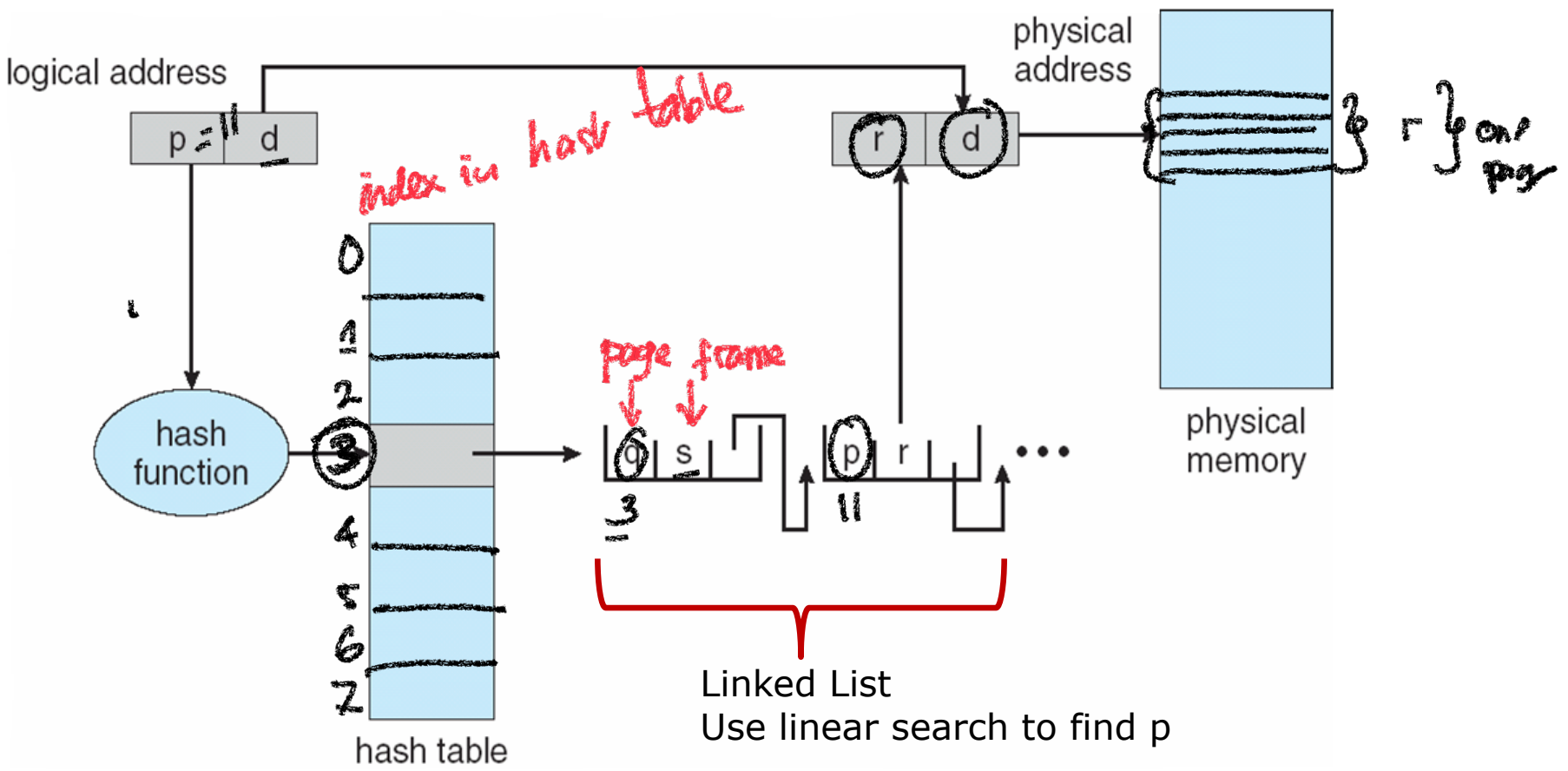
Three-level Paging Scheme



Hashed Page Tables

- Common in address spaces > 32 bits
- The virtual page number is hashed into a page table
  - This page table contains a chain of elements hashing to the same location
- Each element contains (1) the virtual page number (2) the value of the mapped page frame (3) a pointer to the next element
- Virtual page numbers are compared in this chain searching for a match
  - If a match is found, the corresponding physical frame is extracted
- Variation for 64-bit addresses is **clustered page tables**
  - Similar to hashed but each entry refers to several pages (such as 16) rather than 1
  - Especially useful for **sparse** address spaces (where memory references are non-contiguous and scattered)

Hashed Page Table



e.g.
m = 8 bits
n = 4 bits offset ⇝ page size = 2^4 = 16 Bytes

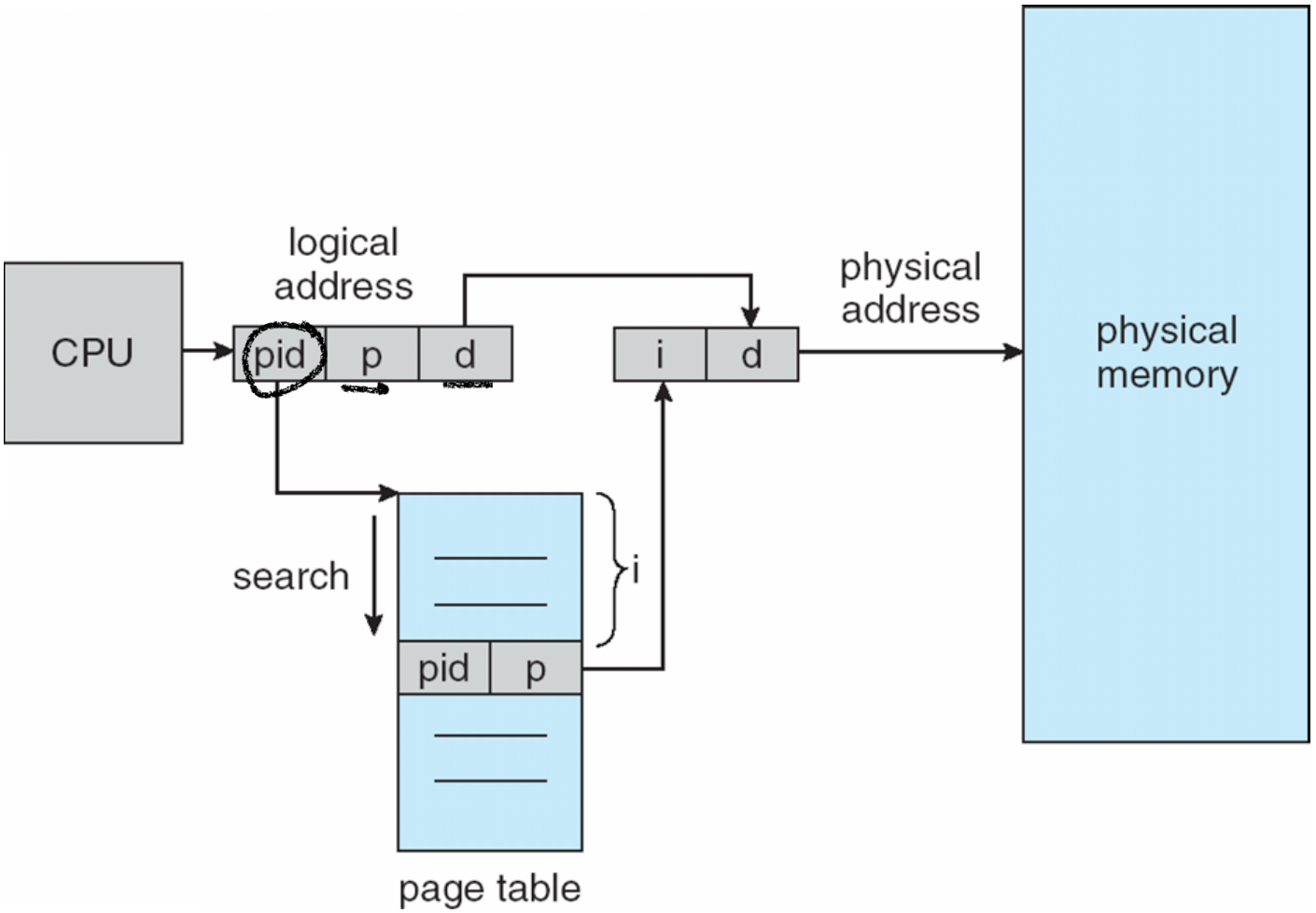
hash(p) = p mod 8
= 3 mod 8 = 3
11 mod 8 = 3

2^4 ⇝ # entries in page table

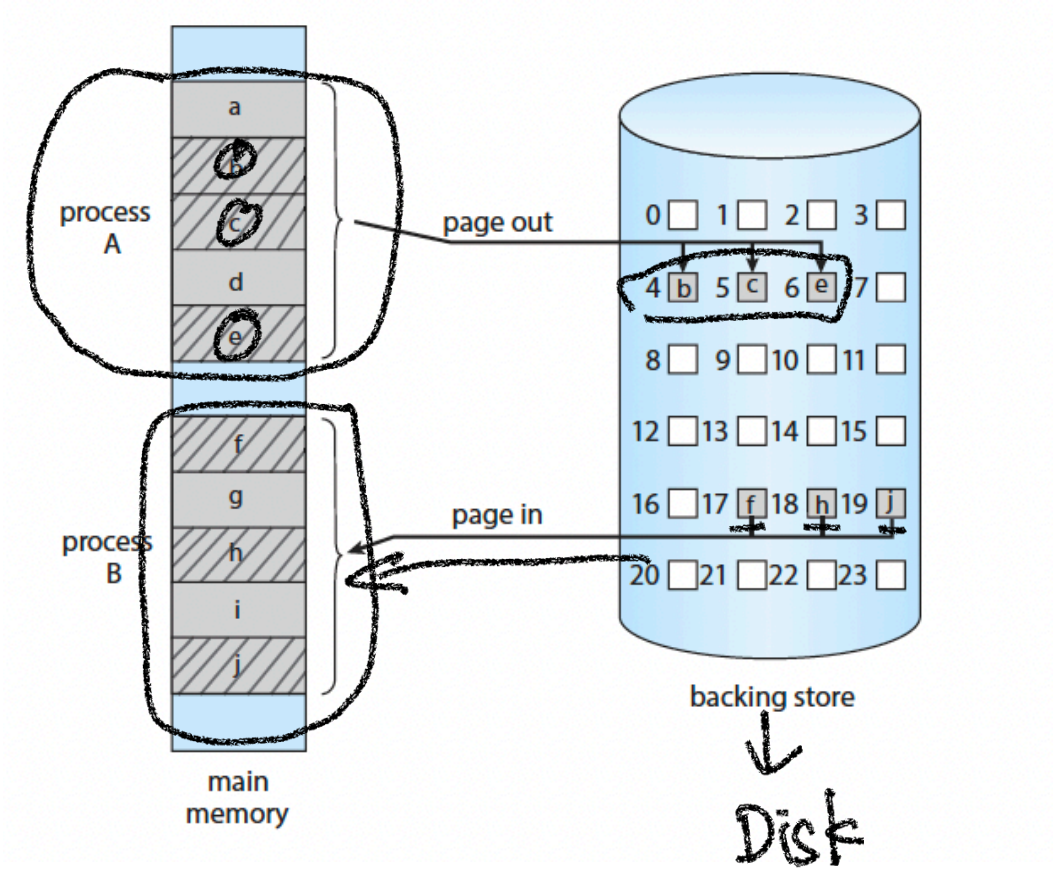
Inverted Page Table

- 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



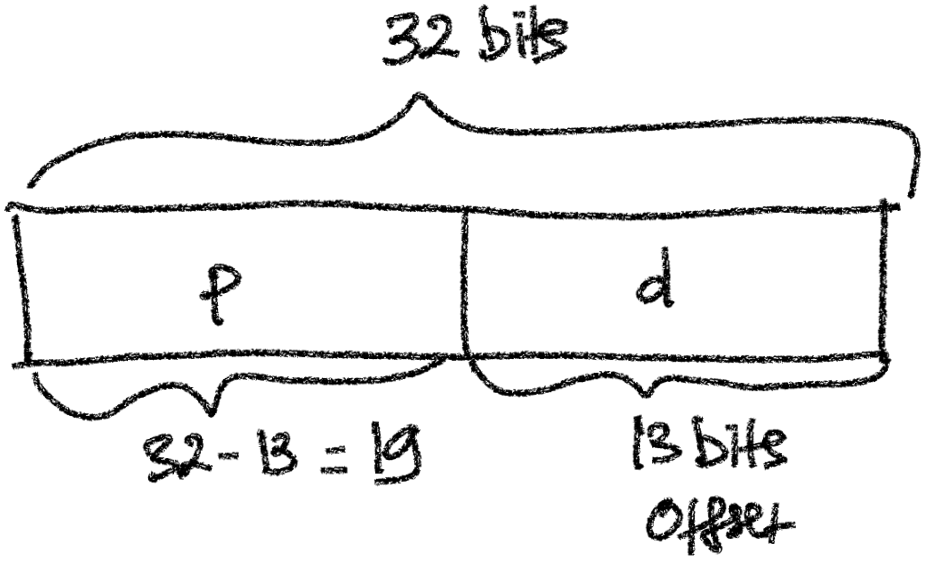
Swapping with Paging



Practice TopHat 9.1

1 KB = 2^{10}

A 32-bit single-level paging system with an 8 KB page size has _____ entries in the page table.



8 KB = $2^{?}$ Bytes

$2^3 \times 2^{10}$ Bytes = $2^{3+10} = 2^{13}$

$2^0 = 1$

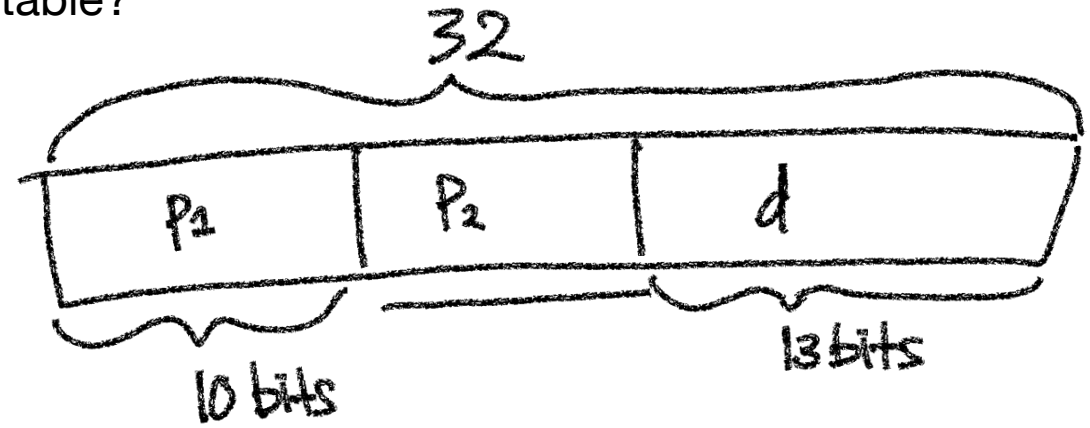
$2^1 = 2$ ×2

$2^2 = 4$ ×2

$2^3 = 8$

Practice TopHat 9.2

Consider a 32-bit address for a two-level paging system with an 8 KB page size. The outer page table has 1024 entries. How many bits are used to represent the second-level page table?



$8KB = 2^{13}$ Bytes

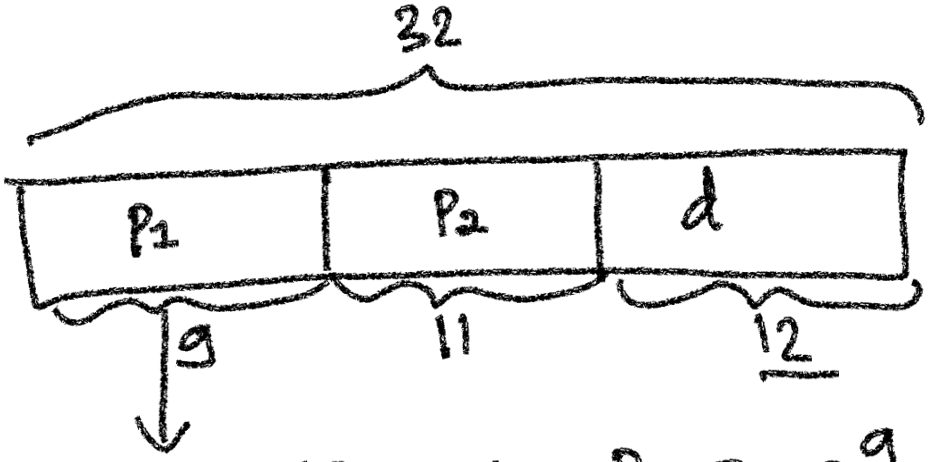
1024 entries = 1024 Bytes

1 KB = 1024 Bytes

2^{10} Bytes

Practice TopHat 9.3

A two-level paging system with 32 bit addresses has 512 entries for the outer page table and 2048 entries for the inner page table. What is the size of a page?



$2^{10} = \frac{1024 \times 2}{2}$

$2^9 = 512$

1 KB = 2^{10} = 1024

1 MB = 2^{20}

1 GB = 2^{30}

512 entries = 512 B ≈ 2^9 B

2048 entries = $\frac{2048\ B}{4 \times 2} \approx 2^{11}$

d = 12

Page size = 2^{12} Bytes = 4096

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

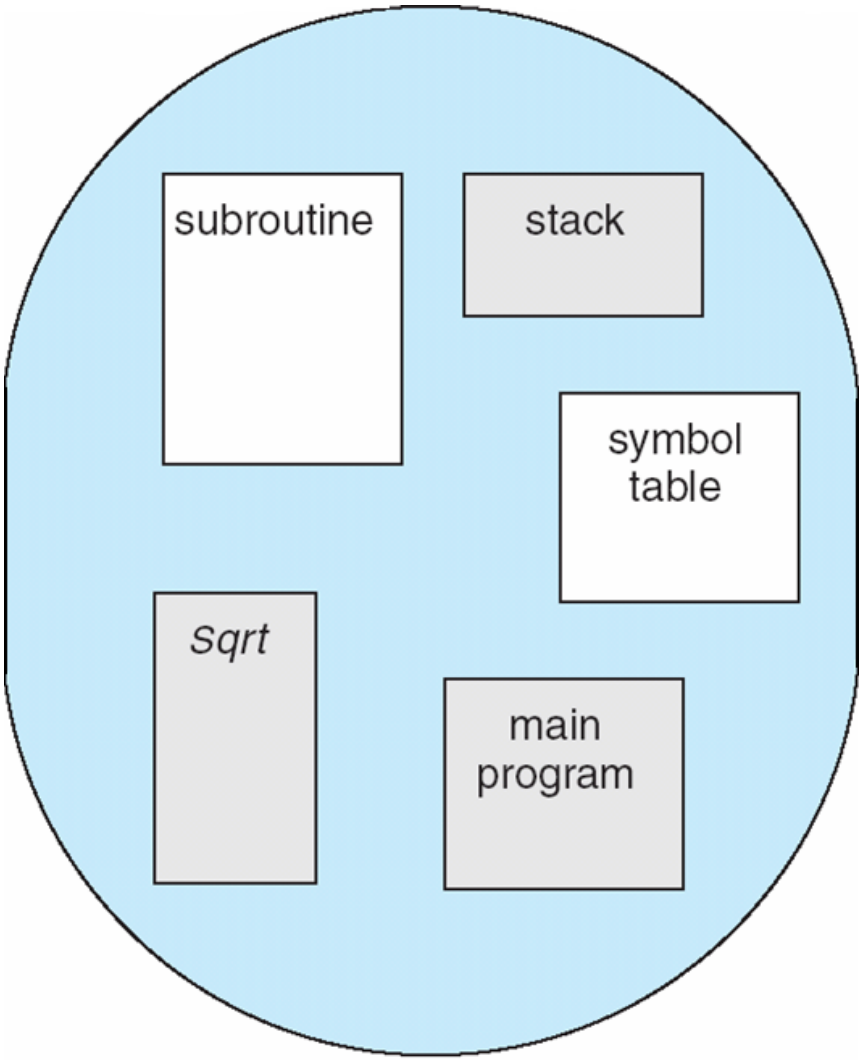
    common block

    stack

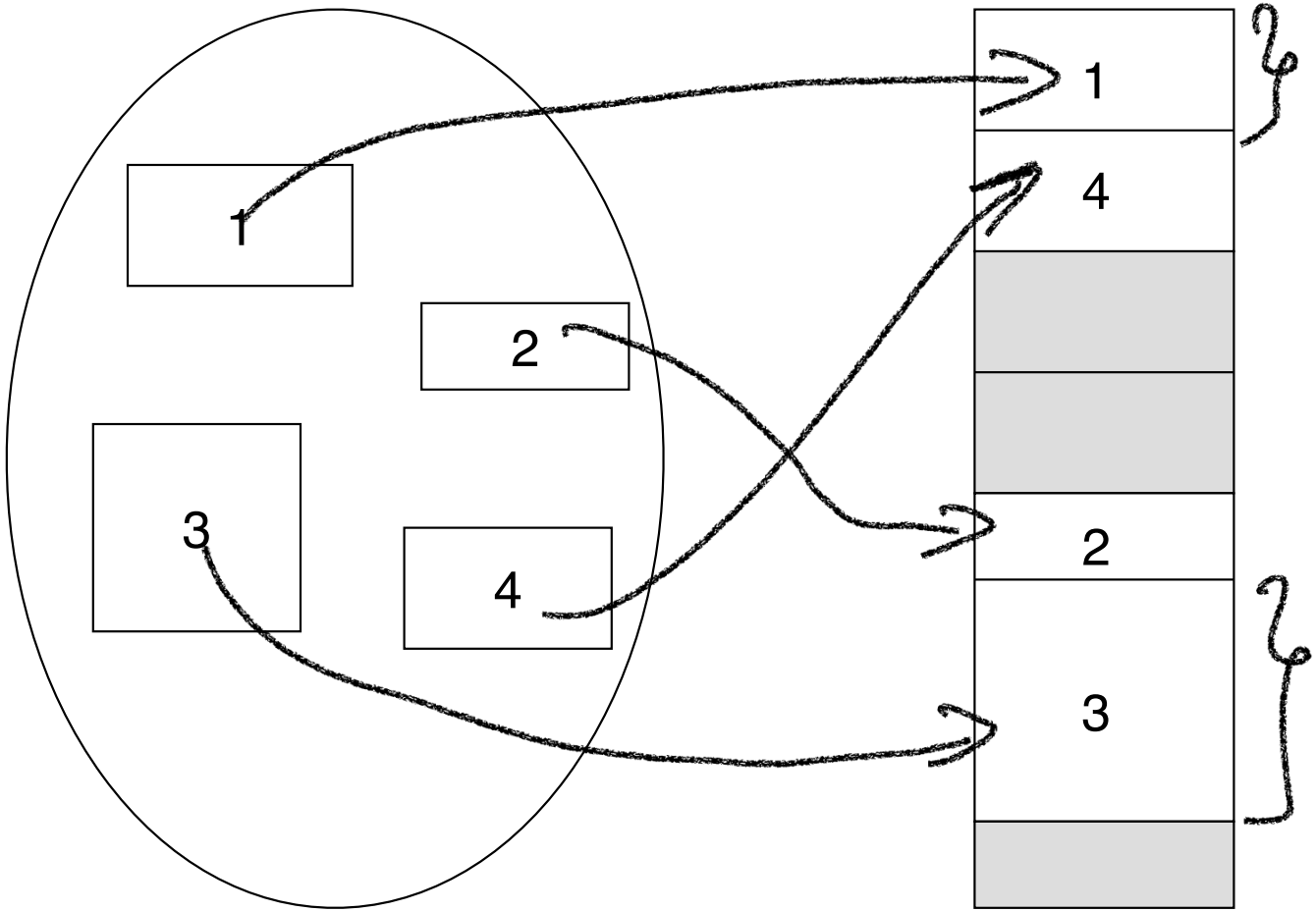
    symbol table

    arrays

User's View of a Program



Logical View of Segmentation



Segmentation Architecture

- Logical address consists of a two tuple:

  <segment-number, offset>,

- **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

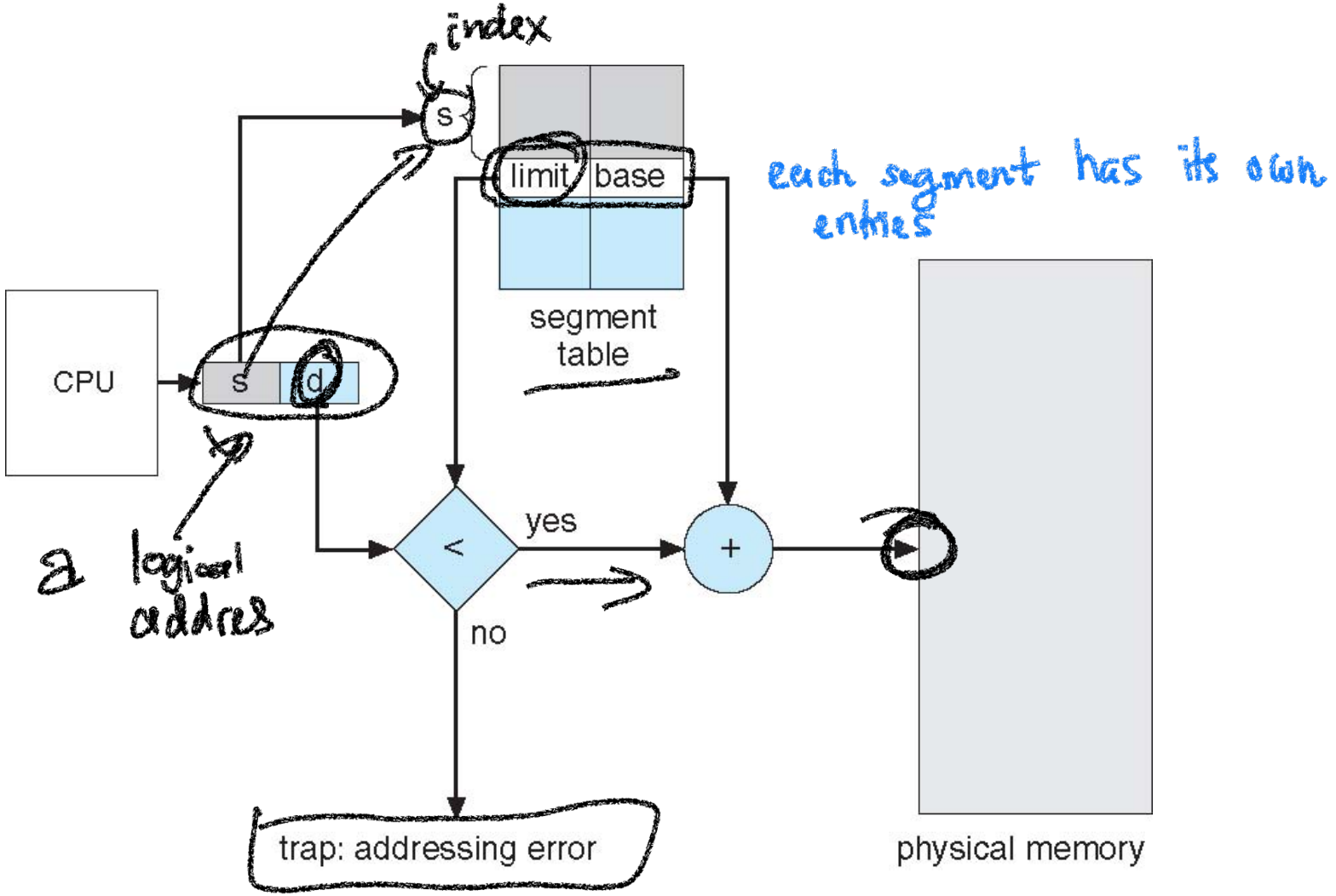
- **Segment-table length register (STLR)** indicates number of segments used by a program;

  segment number ***s*** is legal if ***s*** < **STLR**

Segmentation Architecture (Cont.)

- Protection
  - With each entry in segment table associate:
    - validation bit = 0 ⇒ 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



eg

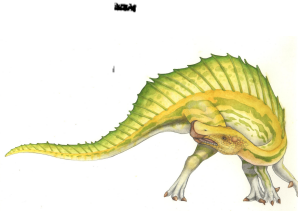
s	d

<3> 80

S=3 in segment table

400	100

limit base



End of Chapter 9