Main Memory

Outline

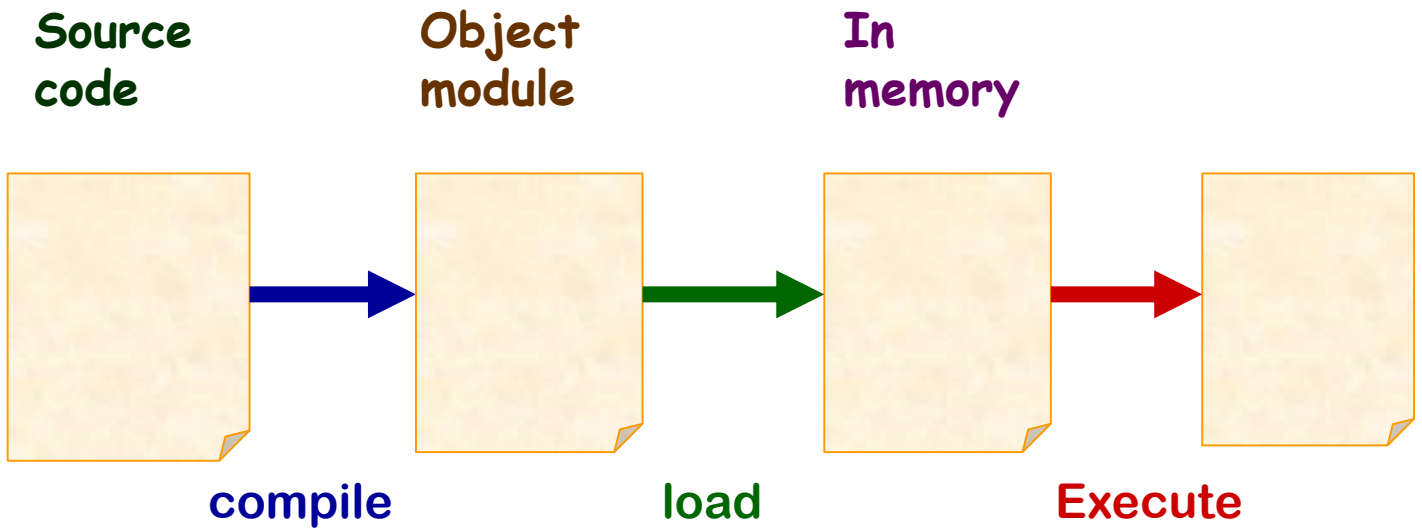
- Background
- Contiguous Memory Allocation
- Paging
- Structure of Page Table
- Segmentation

Background

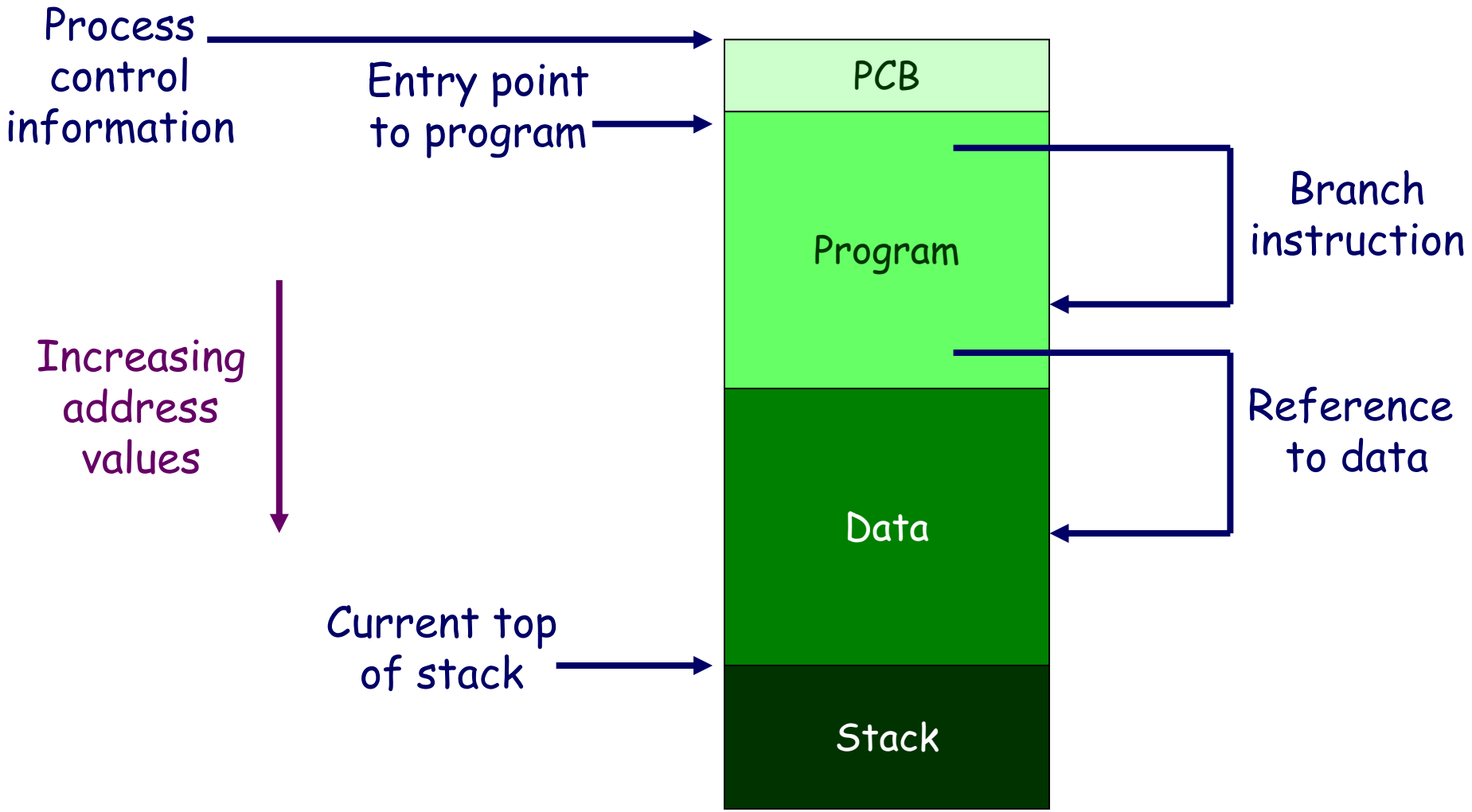
Recall

- Program must be brought (from disk) into memory and placed within a process for it to be run
  - All data in memory before and after processing
  - All instructions in memory in order to execute

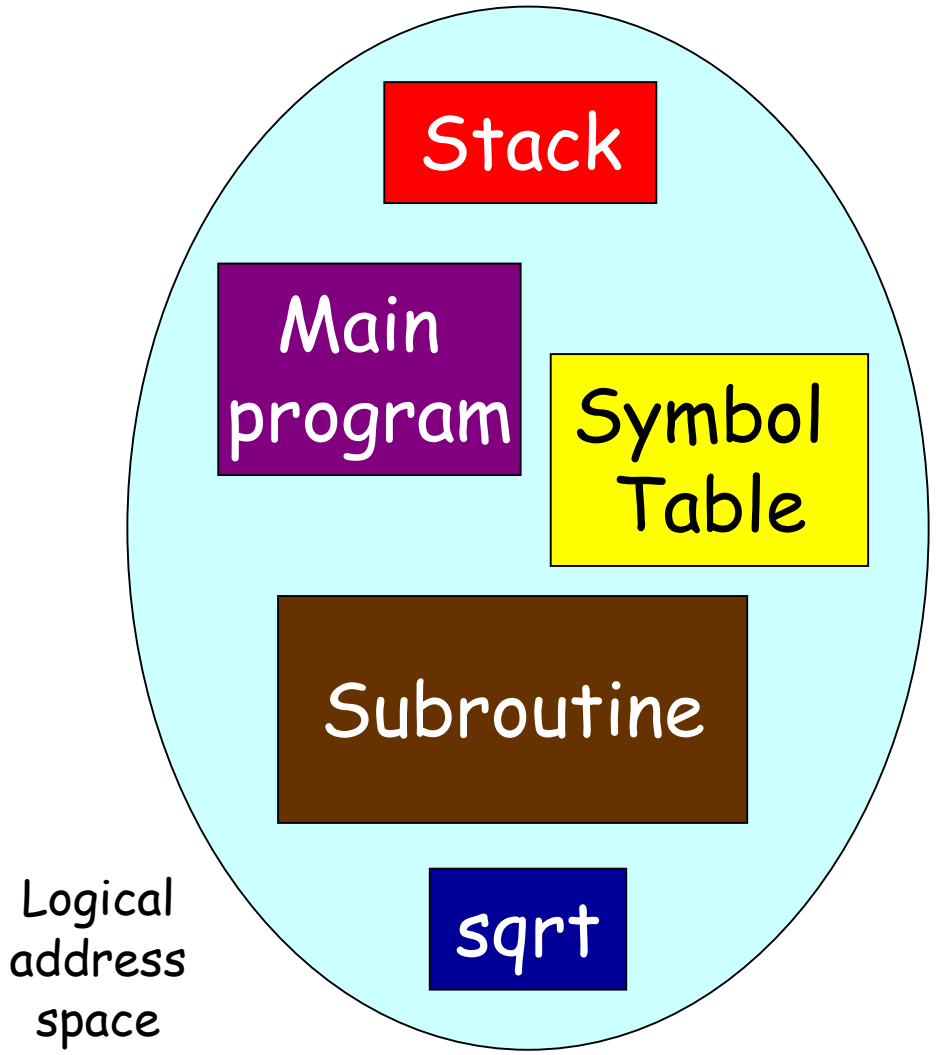
Background



Process in Memory



User's View of a Program

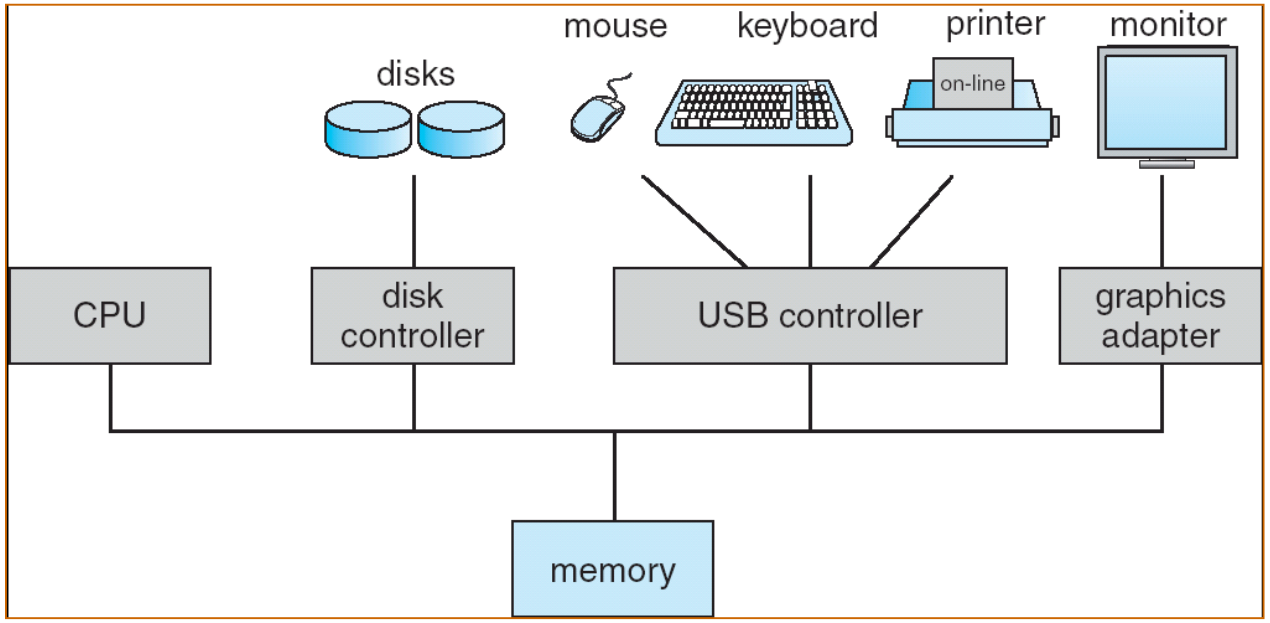


Recall

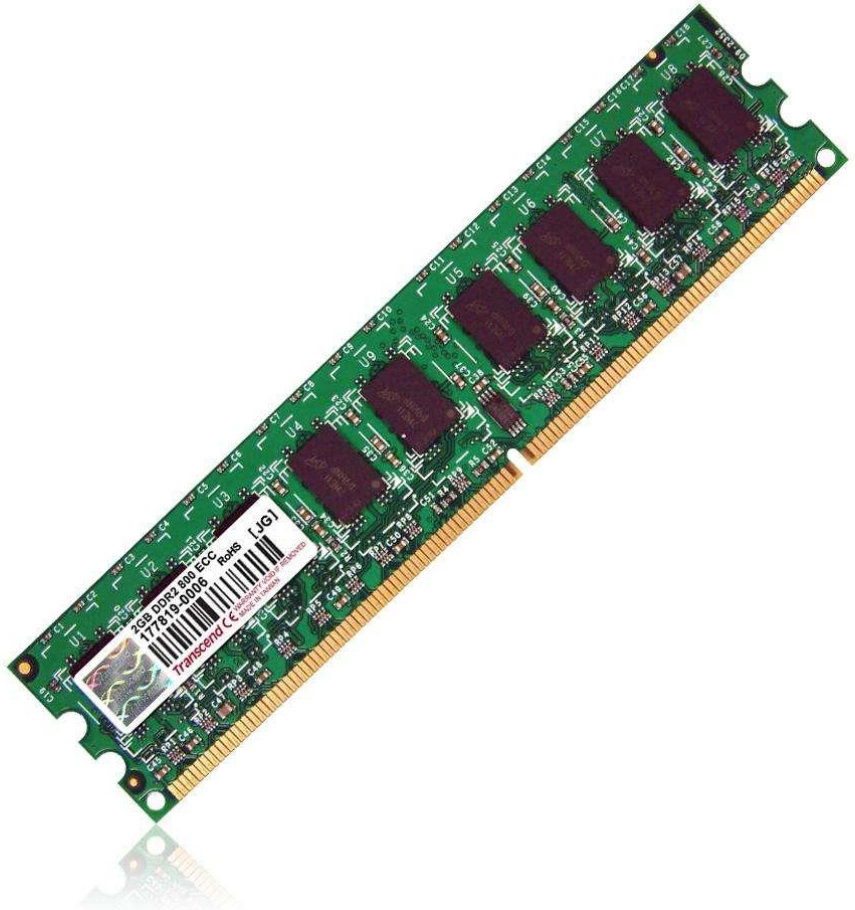
- Program must be brought (from disk) into memory and placed within a process for it to be run
  - All data in memory before and after processing
  - All instructions in memory in order to execute
- Main memory and registers are the **only** storage which CPU can access directly

Computer System Organization

- Computer-system operation
  - One or more CPUs, device controllers connect through common bus providing access to shared memory
  - Concurrent execution of CPUs and devices competing for memory cycles



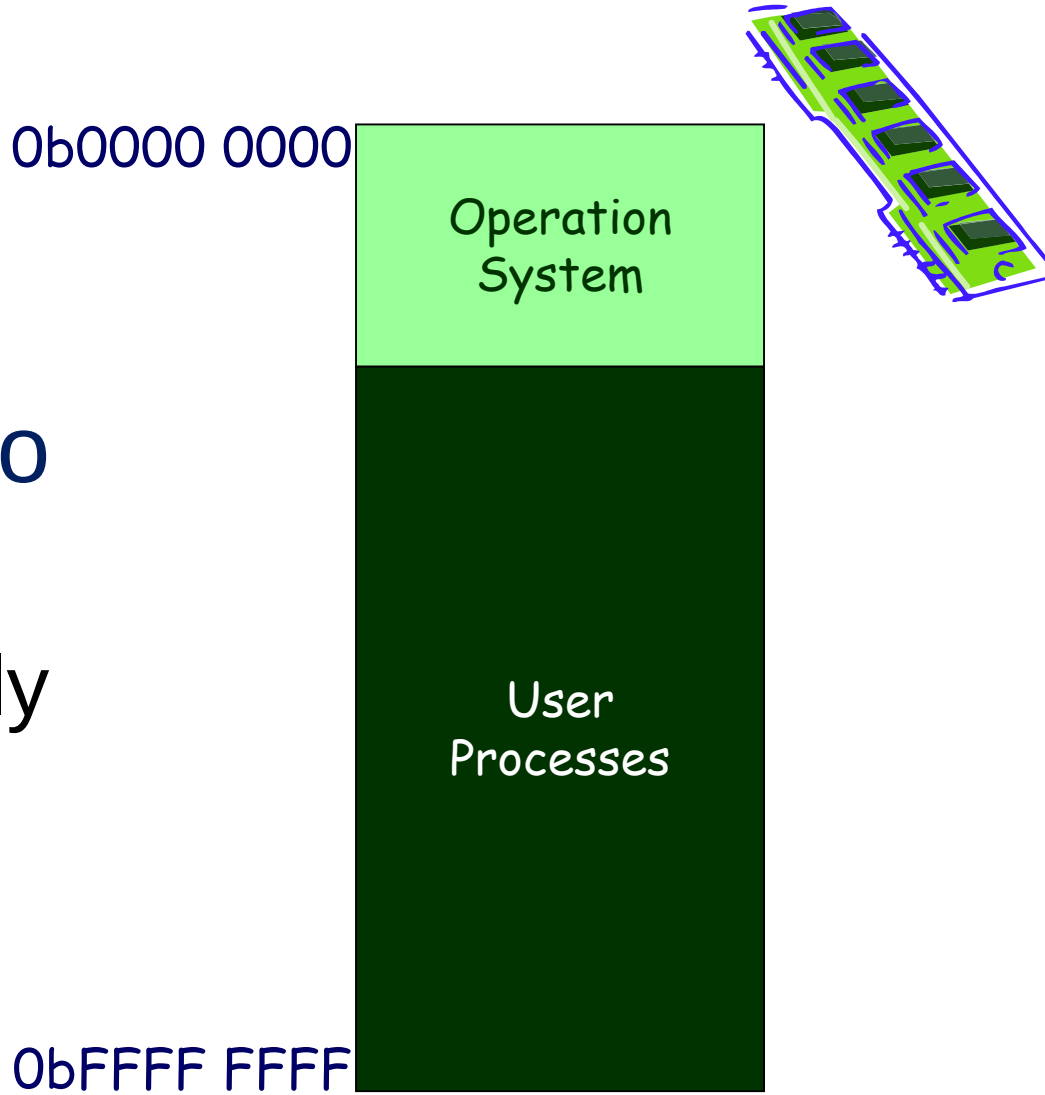
Main memory


2GB DDR2 800 ECC
177819-0006
Transcend
RoHS
[JG]

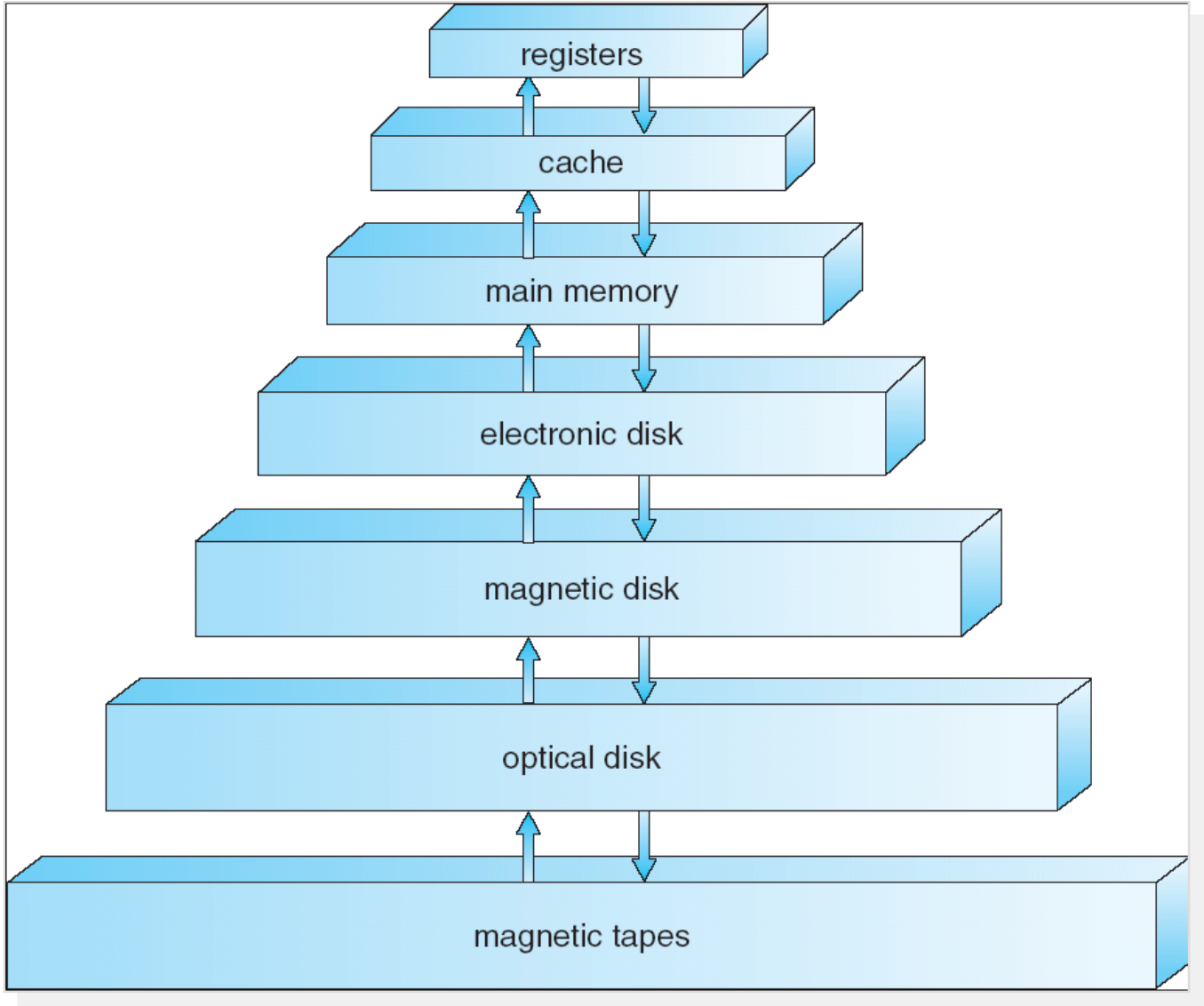

将内存条引脚
上的缺口对准内存
插槽内的凸起
AD72 REV.A

Main memory

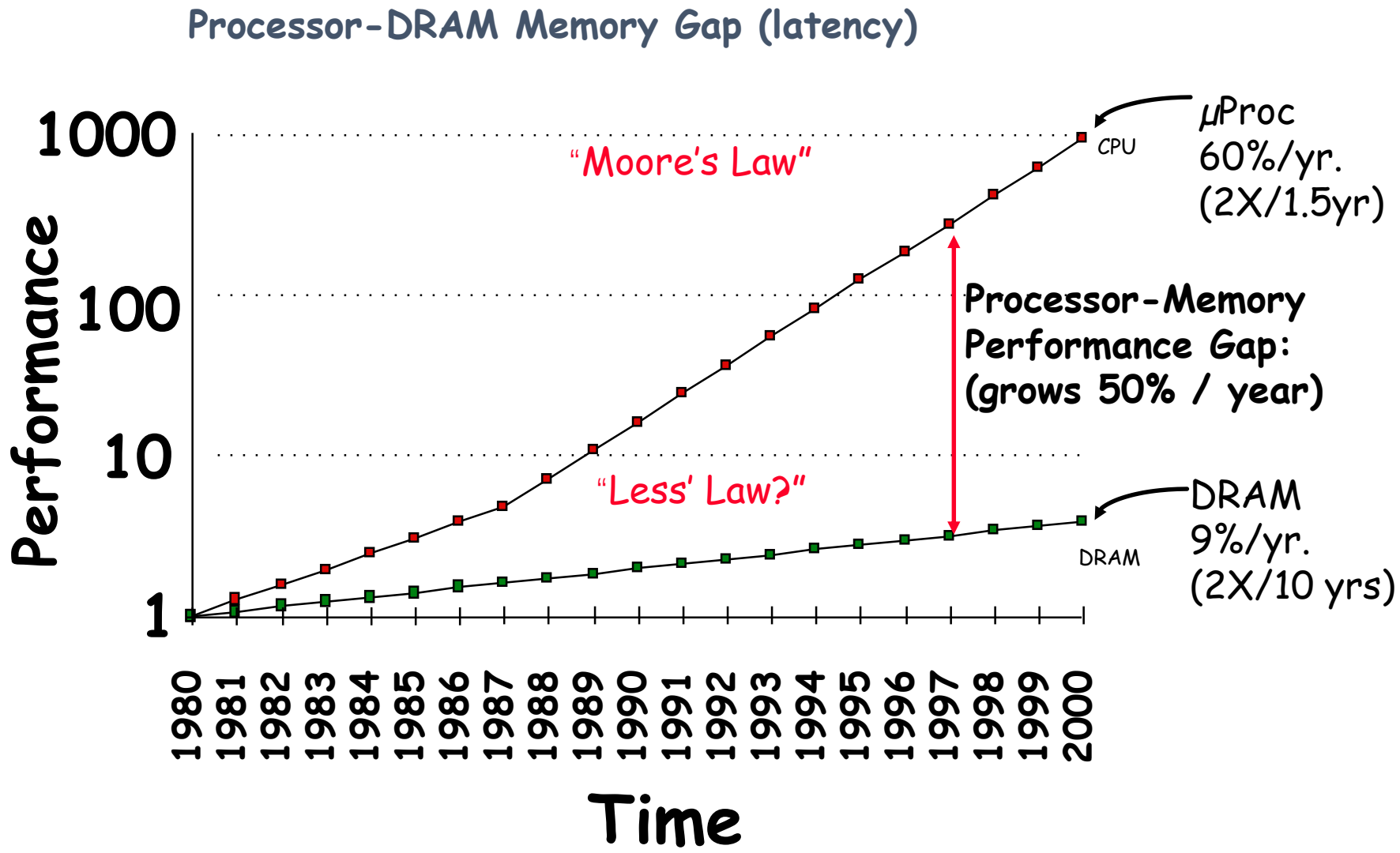
- Address Space:
  - All the addresses and state a process can touch
- Main memory usually divided into two partitions:
  - Resident operating system, usually held in low memory
  - User processes then held in high memory



Storage-Device Hierarchy



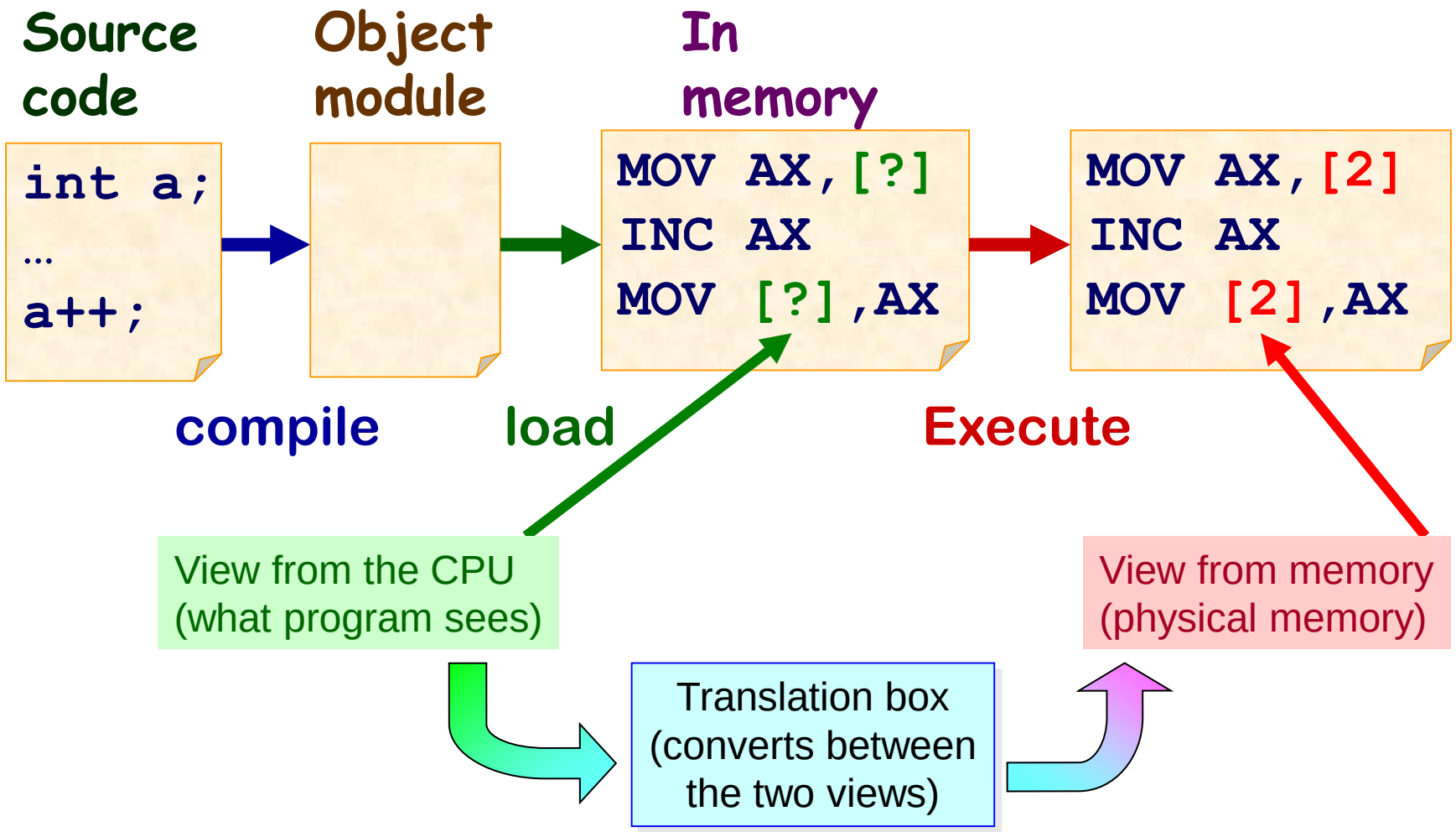
Memory


Processor-DRAM Memory Gap (latency)
Performance
1000
100
10
1
"Moore's Law"
"Less' Law?"
CPU
DRAM
µProc
60%/yr.
(2X/1.5yr)
Processor-Memory
Performance Gap:
(grows 50% / year)
DRAM
9%/yr.
(2X/10 yrs)
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
Time

Memory Management

- Memory management activities
  - Keeping track of which parts of memory are currently being used and by whom
  - Deciding which processes (or parts thereof) and data to move into and out of memory
  - Allocating and deallocating memory space as needed

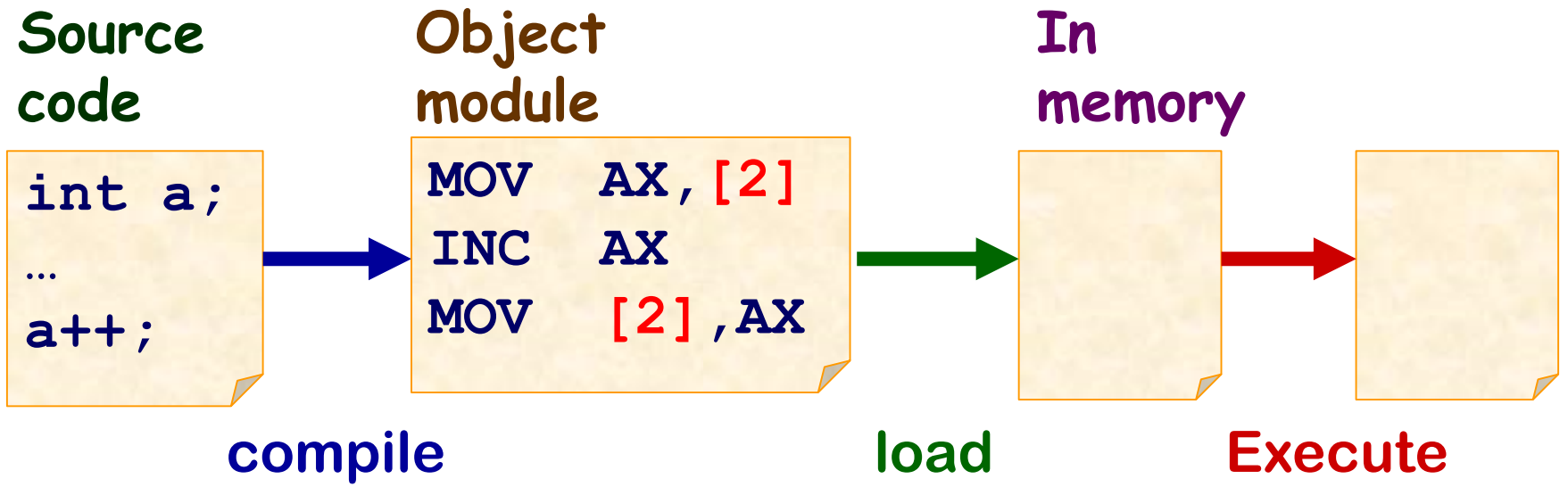
Two Views of Memory



Logical vs. Physical Address Space

- Logical address – generated by the CPU
- 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 and physical addresses differ in execution-time address-binding scheme

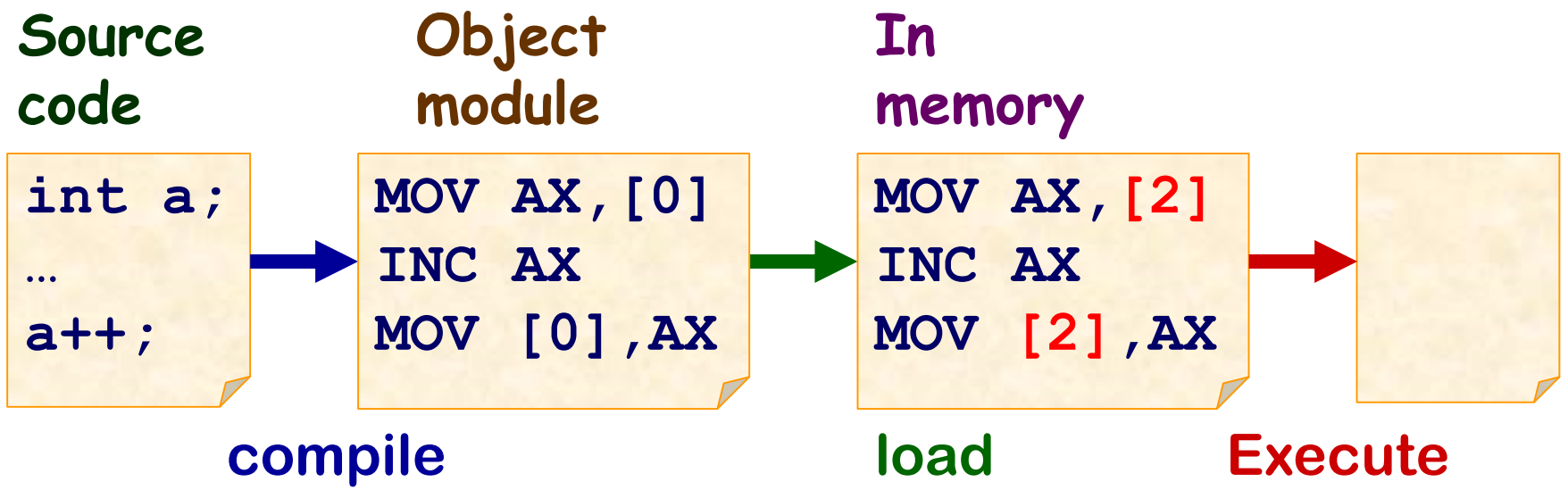
Address binding scheme: in compile time



- Address binding of instructions and data to memory addresses happen at:
  - **1. Compile time**: If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes.

Address binding scheme: in load time

- Address binding of instructions and data to memory addresses happen at
  - **2. Load time**: must generate **relocatable code** if memory location is not known at compile time



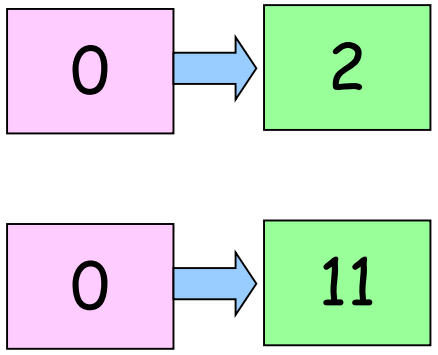
Address binding scheme: in load time

- Example

```
int a;
…
a++;
```

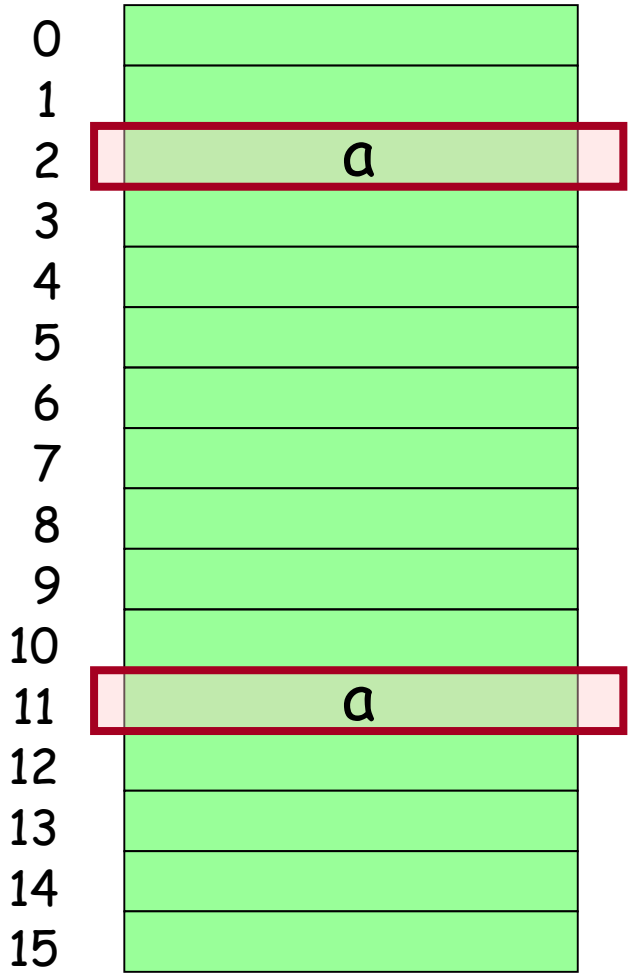
```
MOV   AX, a
INC   AX
MOV   a, AX
```

```
MOV   AX,[0]
INC   AX
MOV   [0],AX
```



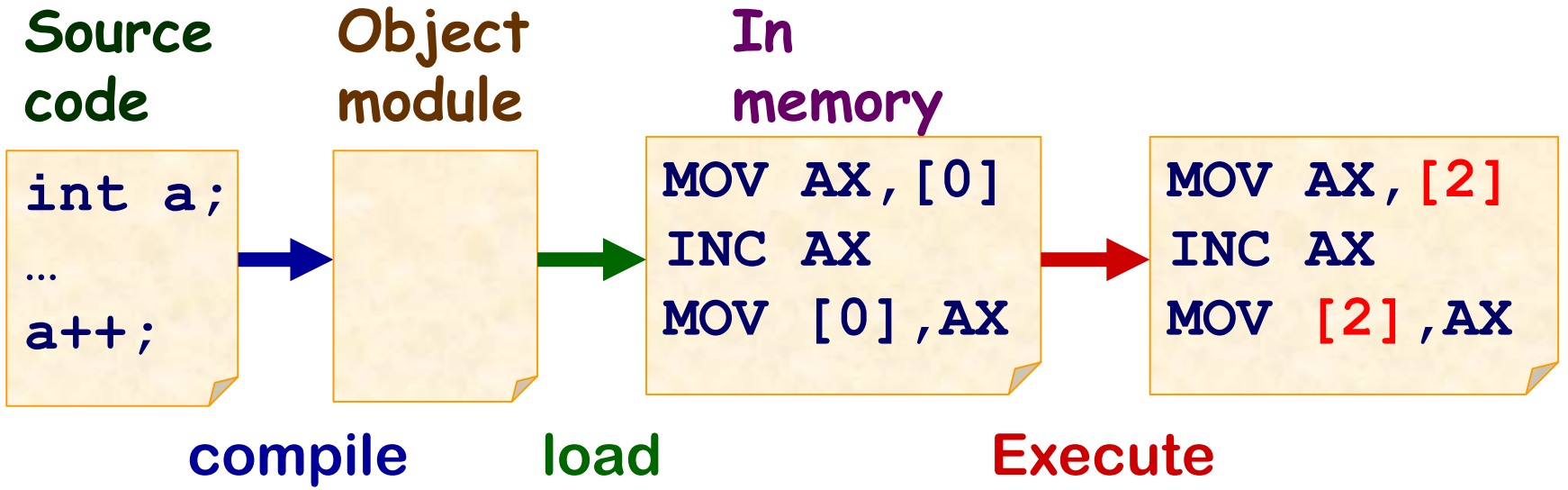

```
MOV   AX,[2]
INC   AX
MOV   [2],AX
```

```
MOV   AX,[11]
INC   AX
MOV   [11],AX
```



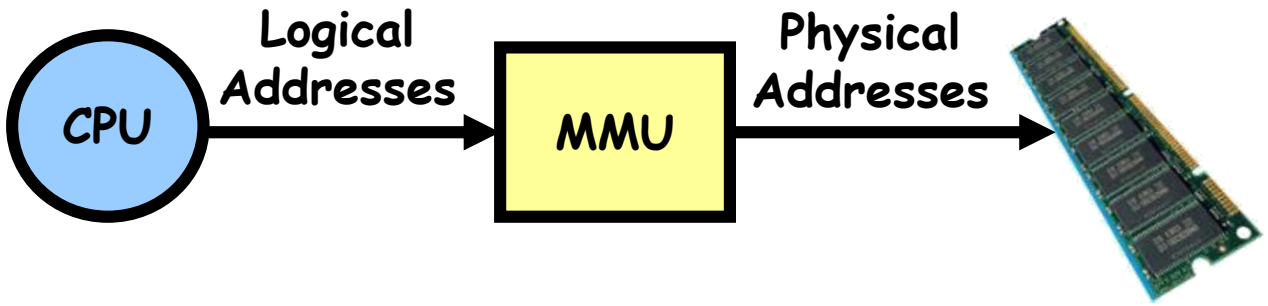
Dynamic Relocation

- Address binding of instructions and data to memory addresses happen at
  - **3. 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)



Memory-Management Unit (MMU)

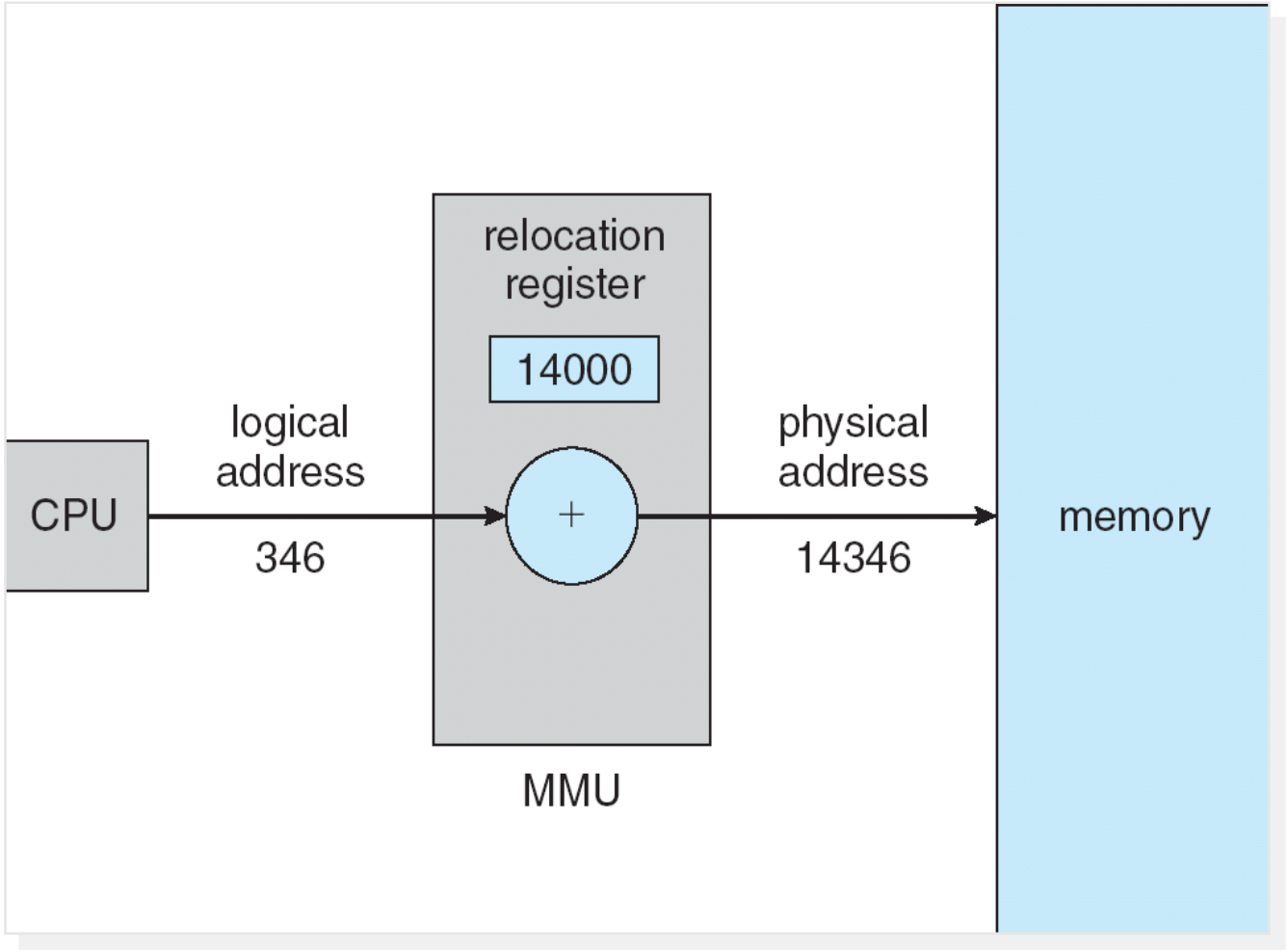
- Hardware device that maps logical address to physical address
- In MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory



- The user program deals with logical addresses; it never sees the real physical addresses

Memory-Management Unit (MMU)

- Dynamic relocation using a relocation register



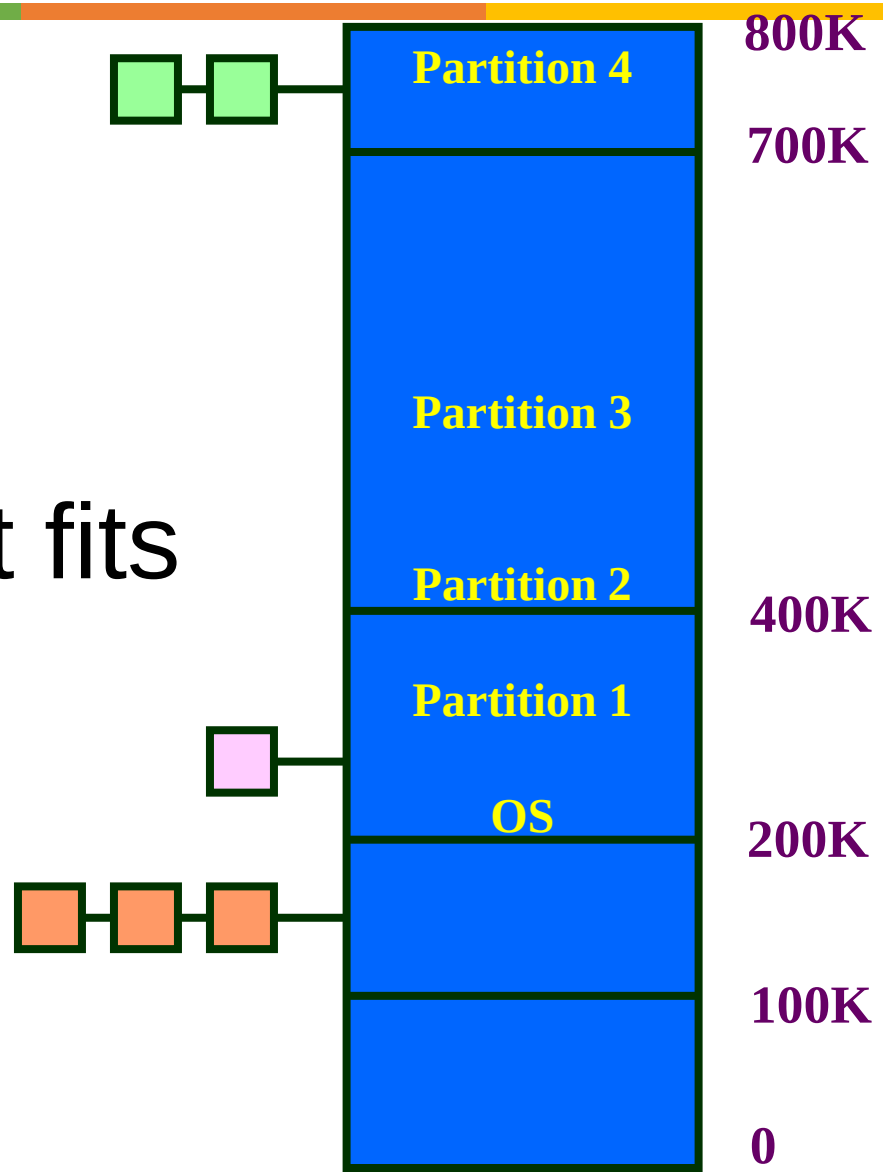
Memory Management Requirement

- Protection
  - Processes should not be able to reference memory locations in another process without permission
- Sharing
  - Allow several processes to access the same portion of memory
  - Better to allow each process access to the same copy of the program rather than have their own separate copy

Contiguous Memory Allocation

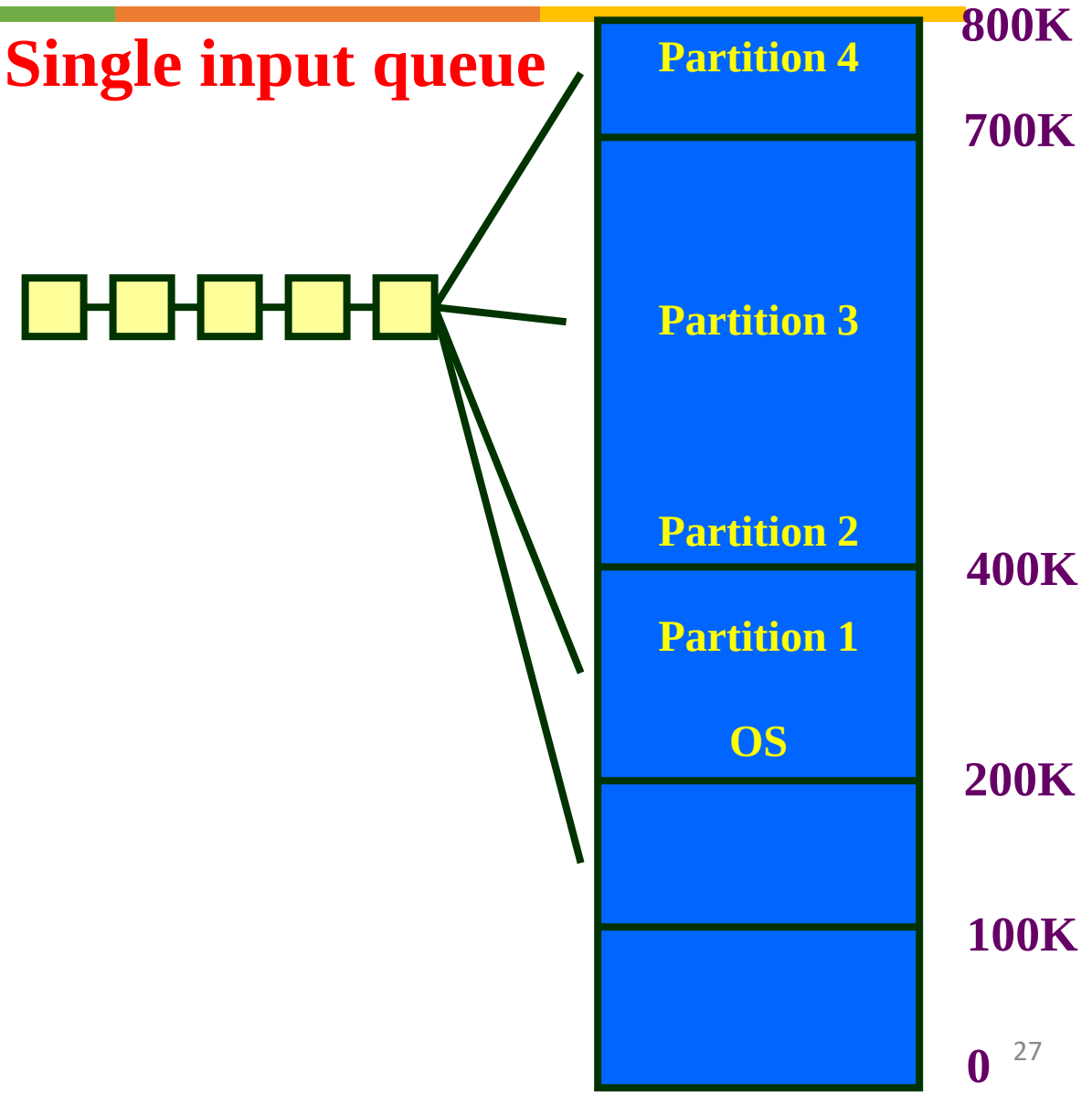
Fixed Partition

- One job each partition
- In multiple queues, each process is assigned to the smallest partition in which it fits and minimizes the internal fragmentation problem.



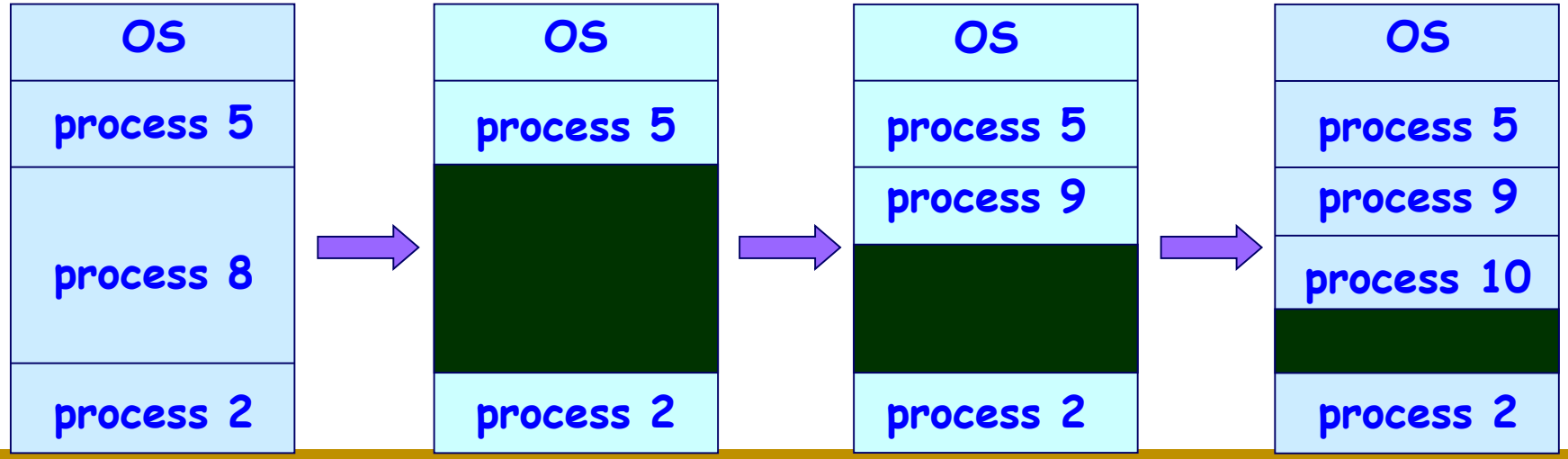
Fixed Partition

- In single queue, the process is assigned to the smallest available partition and the level of multiprogramming is increased.
- Main memory use is inefficient. Any program, no matter how small, occupies an entire partition.

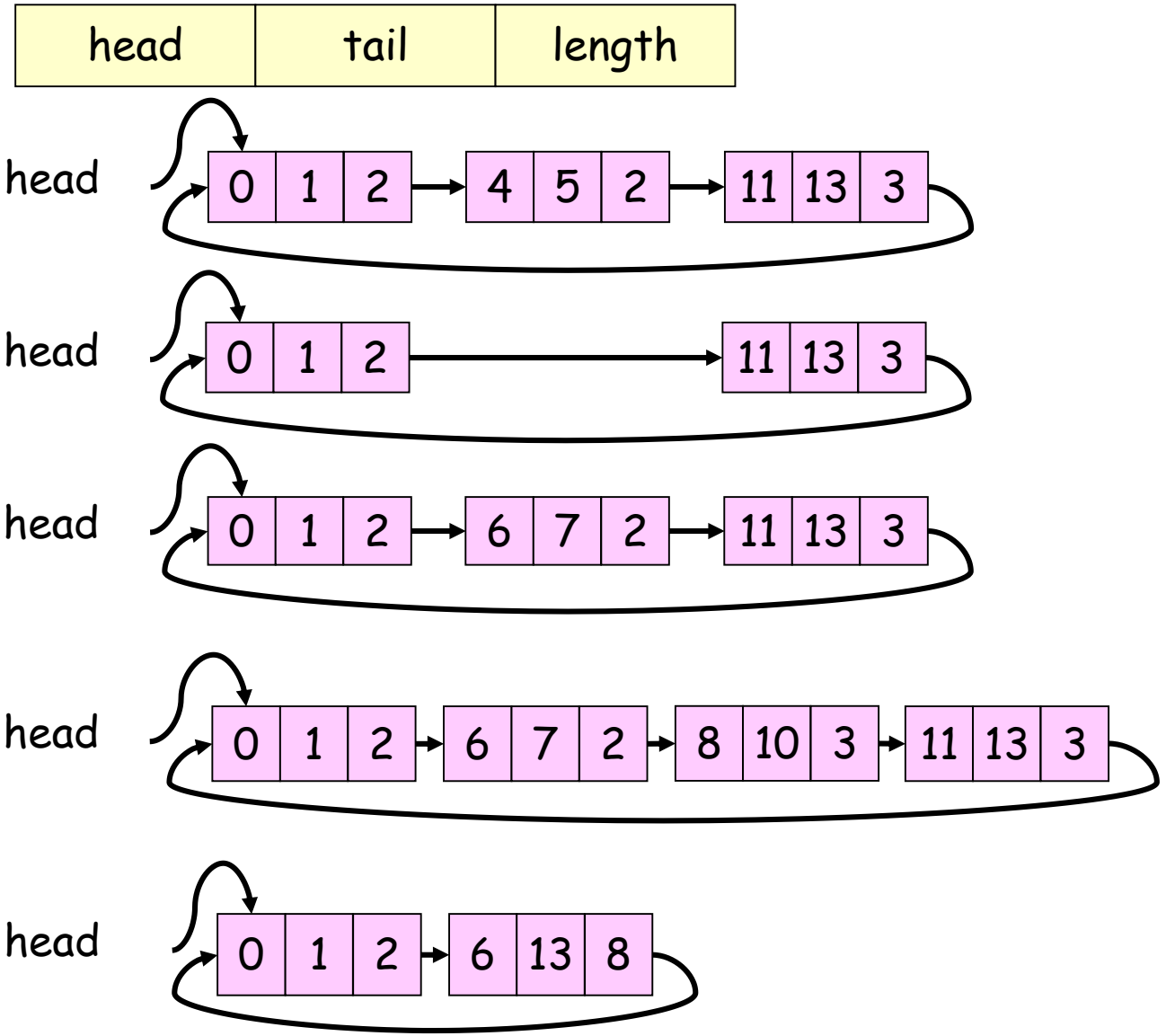


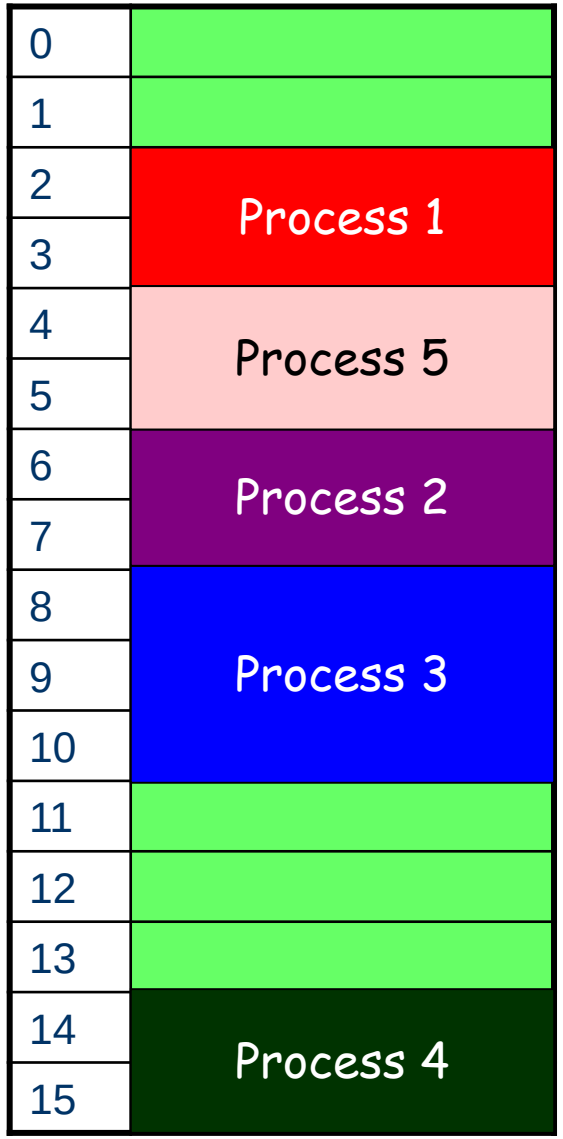
Dynamically Partition

- Multiple-partition allocation
  - 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
  - Operating system maintains information about:
    a) allocated partitions; b) free partitions (hole)



Dynamic Storage-Allocation Problem



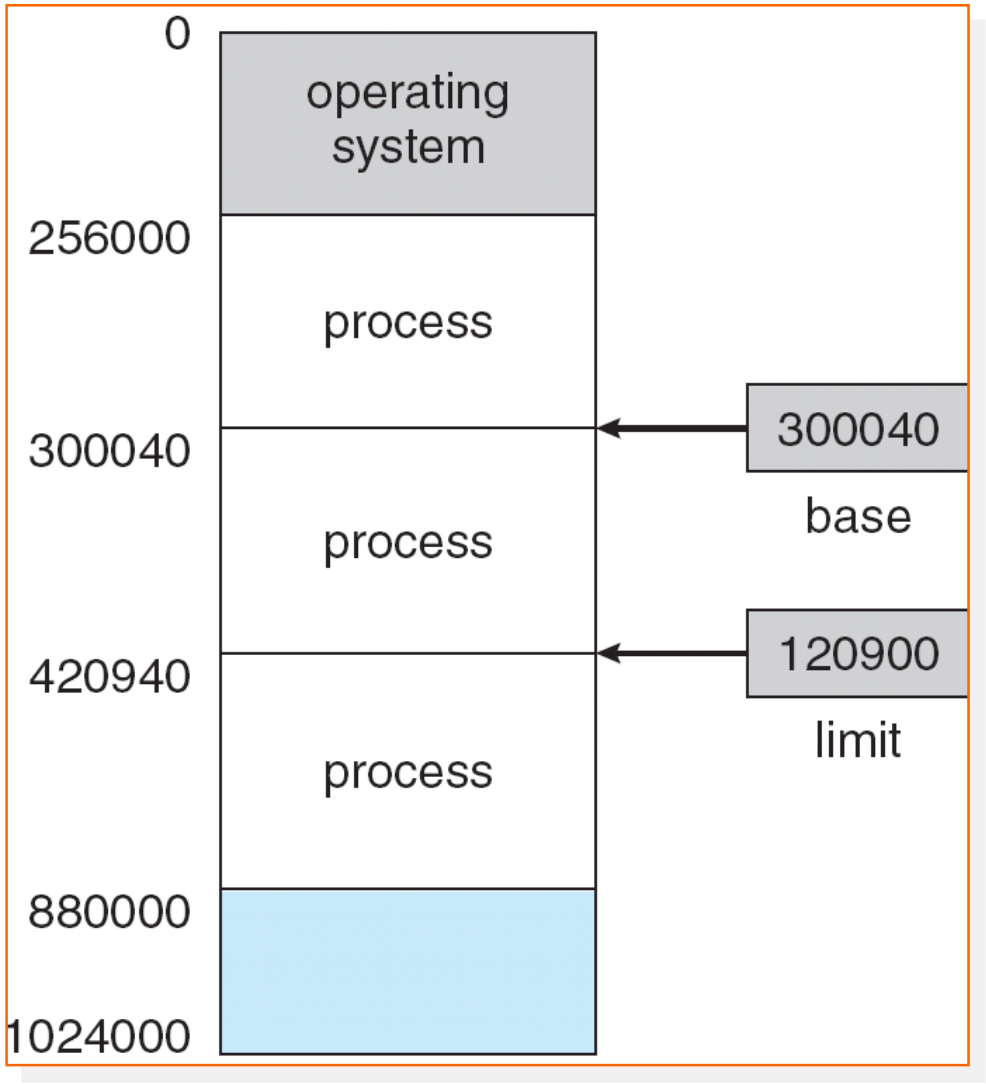


Contiguous Memory Allocation

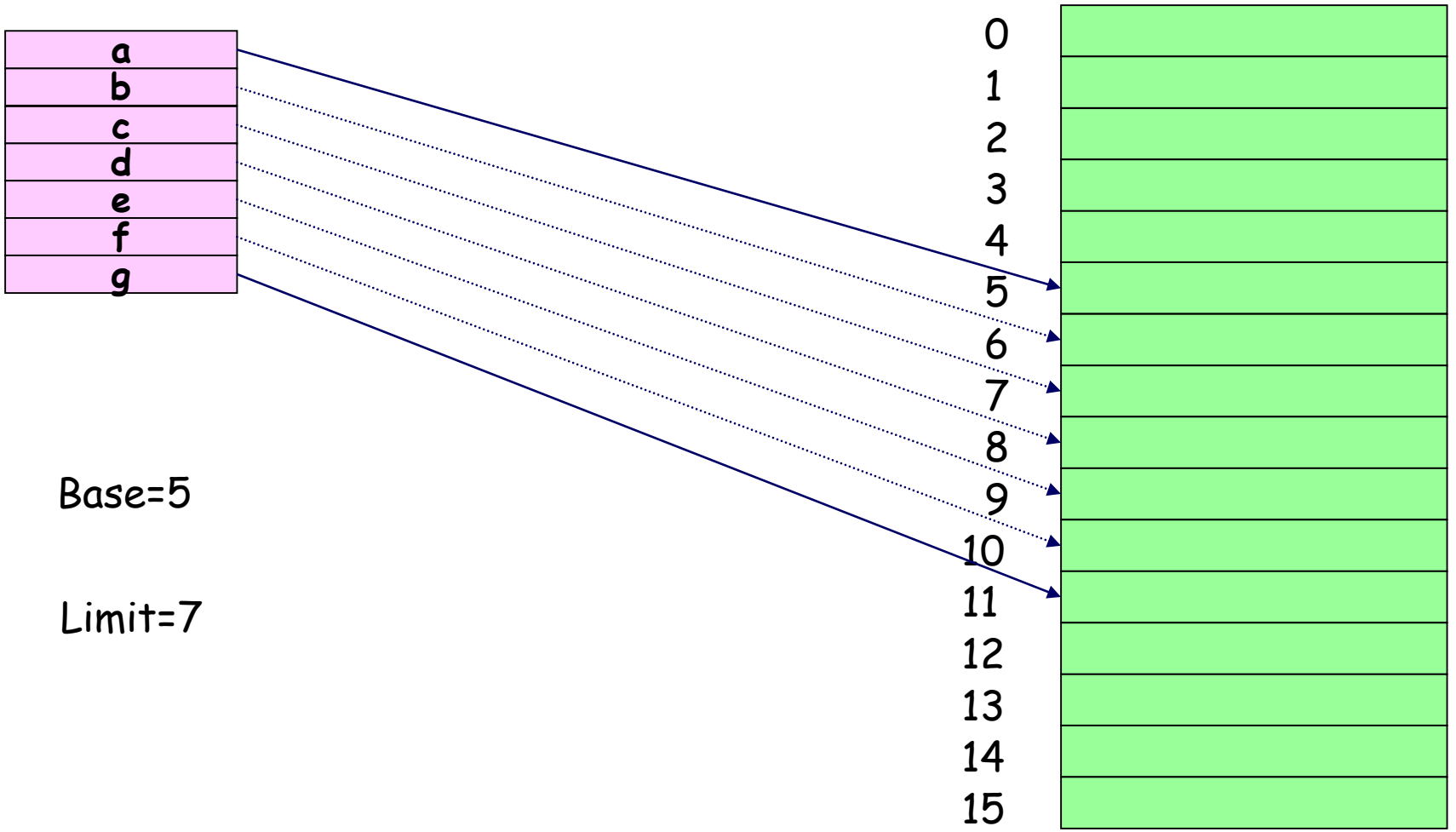
- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - **Base** register contains value of smallest physical address
  - **Limit** register contains range of logical addresses – each logical address must be less than the limit register
  - MMU maps logical address dynamically

Base and Limit Registers

- A pair of base and limit registers define the logical address space

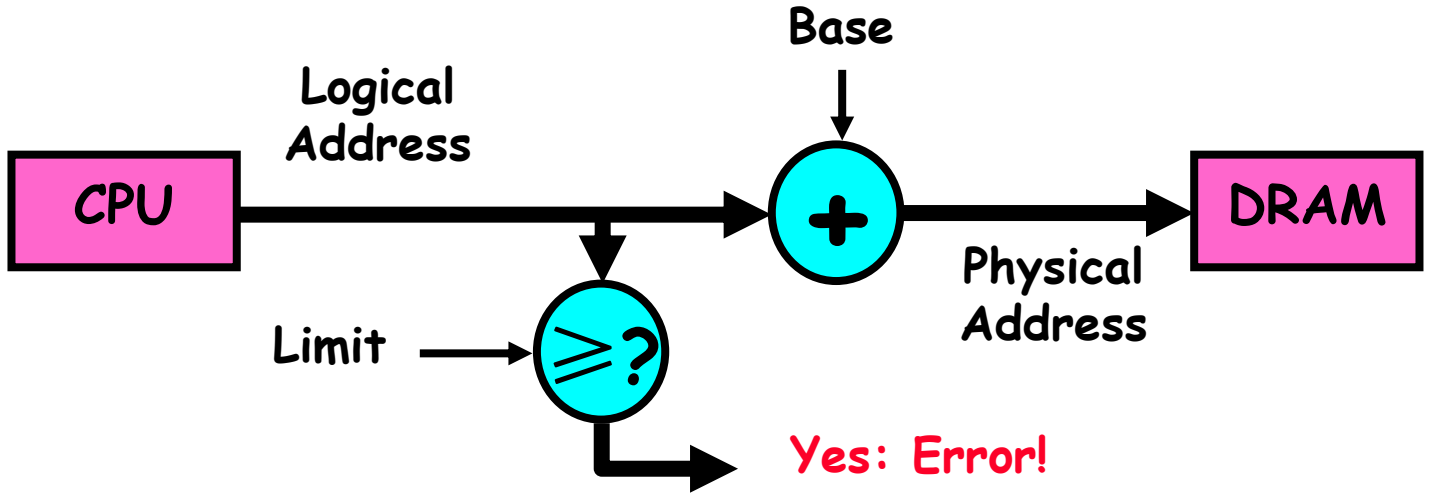


Base and Limit Registers



Base and Limit Registers

- Can use base & bounds/limit for dynamic address translation (Simple form of “segmentation”):
- This gives program the illusion that it is running on its own dedicated machine, with memory starting at 0
  - Program gets continuous region of memory
  - Addresses within program do not have to be relocated when program placed in different region of DRAM

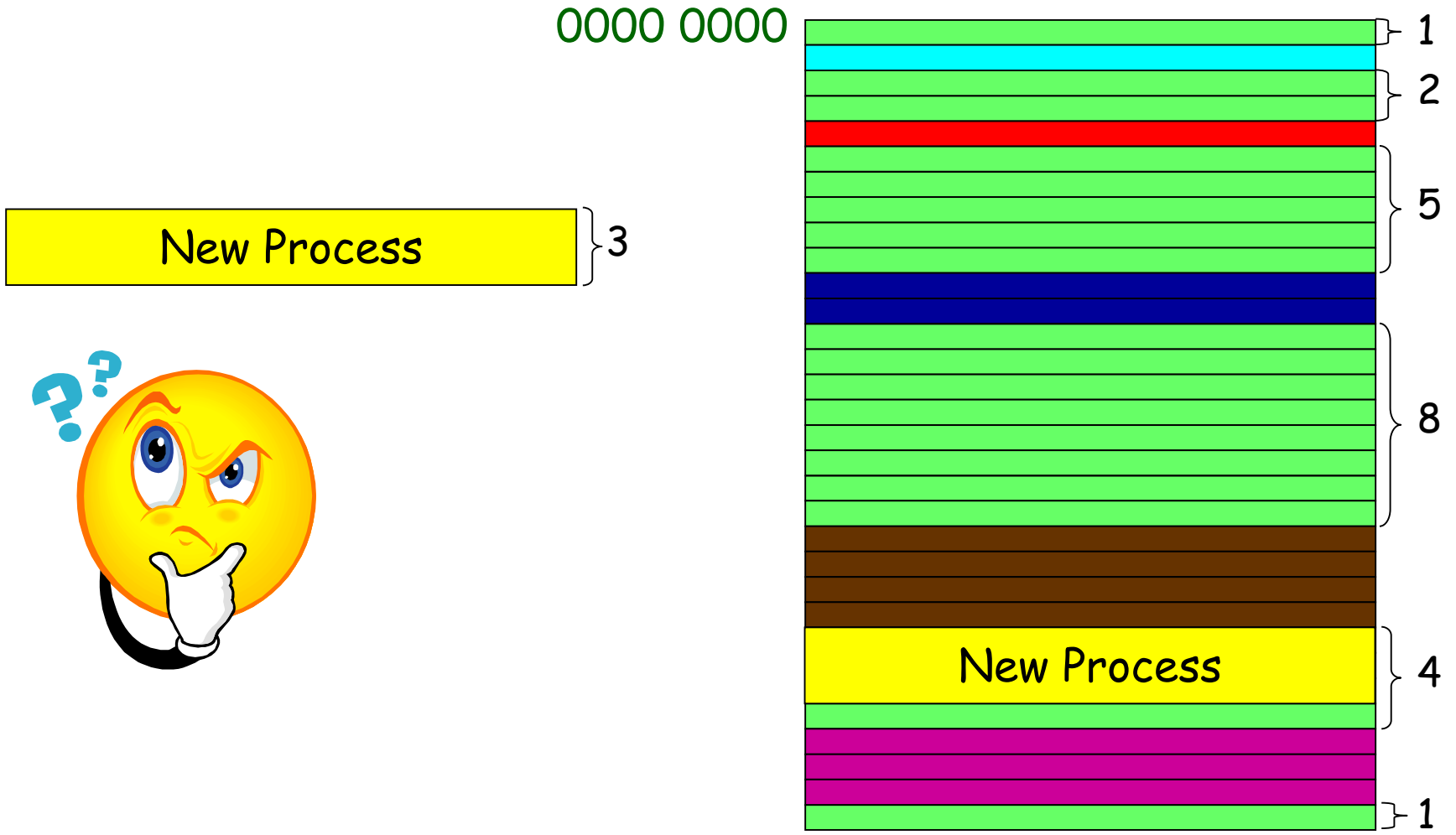


Dynamic Storage-Allocation Problem

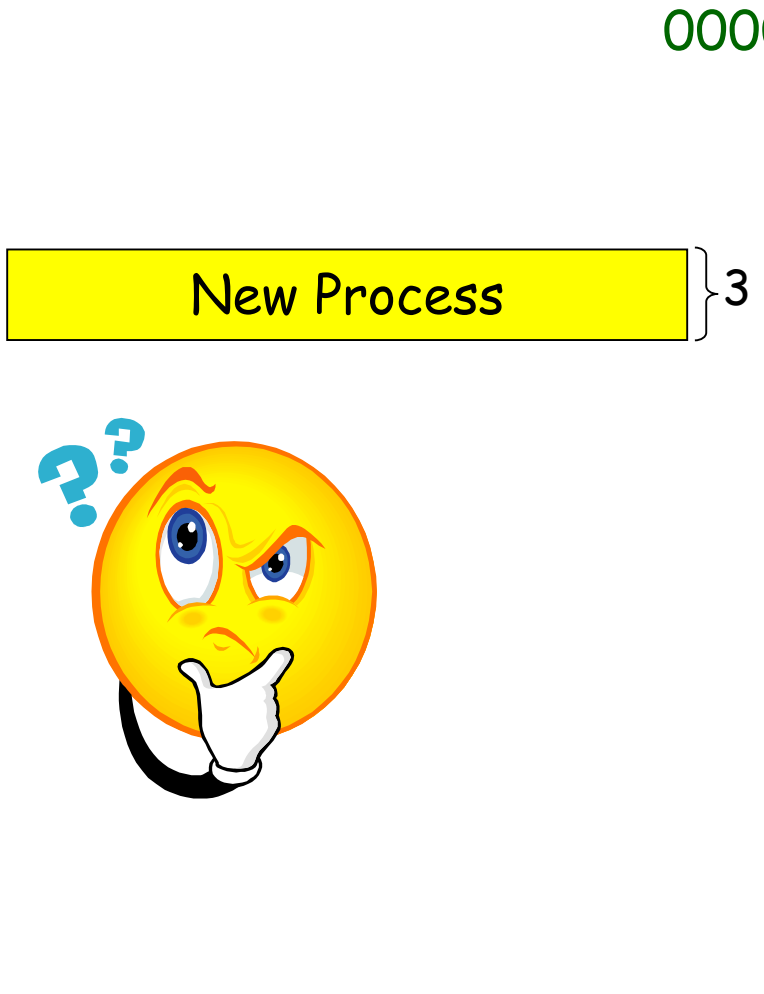
- How to satisfy a request of size ***n*** from a list of free holes
  - **First-fit**: Allocate the **first** hole that is big enough
  - **Best-fit**: Allocate the **smallest** hole that is big enough; must search entire list, unless ordered by size
    - Produces the smallest leftover hole
  - **Worst-fit**: Allocate the largest hole; must also search entire list
    - Produces the largest leftover hole
  - **Next-fit:** Scans memory from the location of the last placement
- First-fit and best-fit better than worst-fit in terms of speed and storage utilization

First-fit

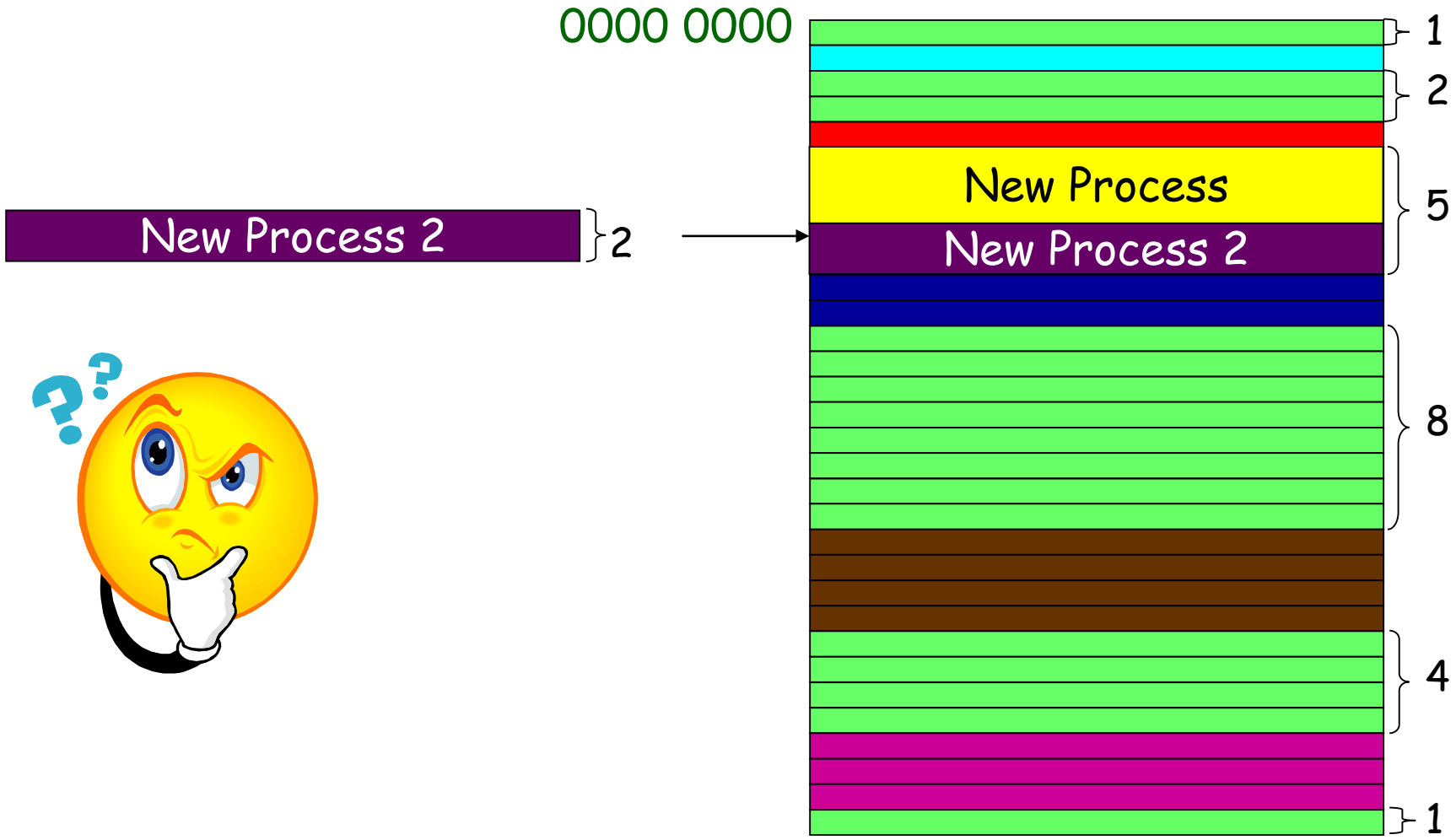
Best-fit



Worst-fit



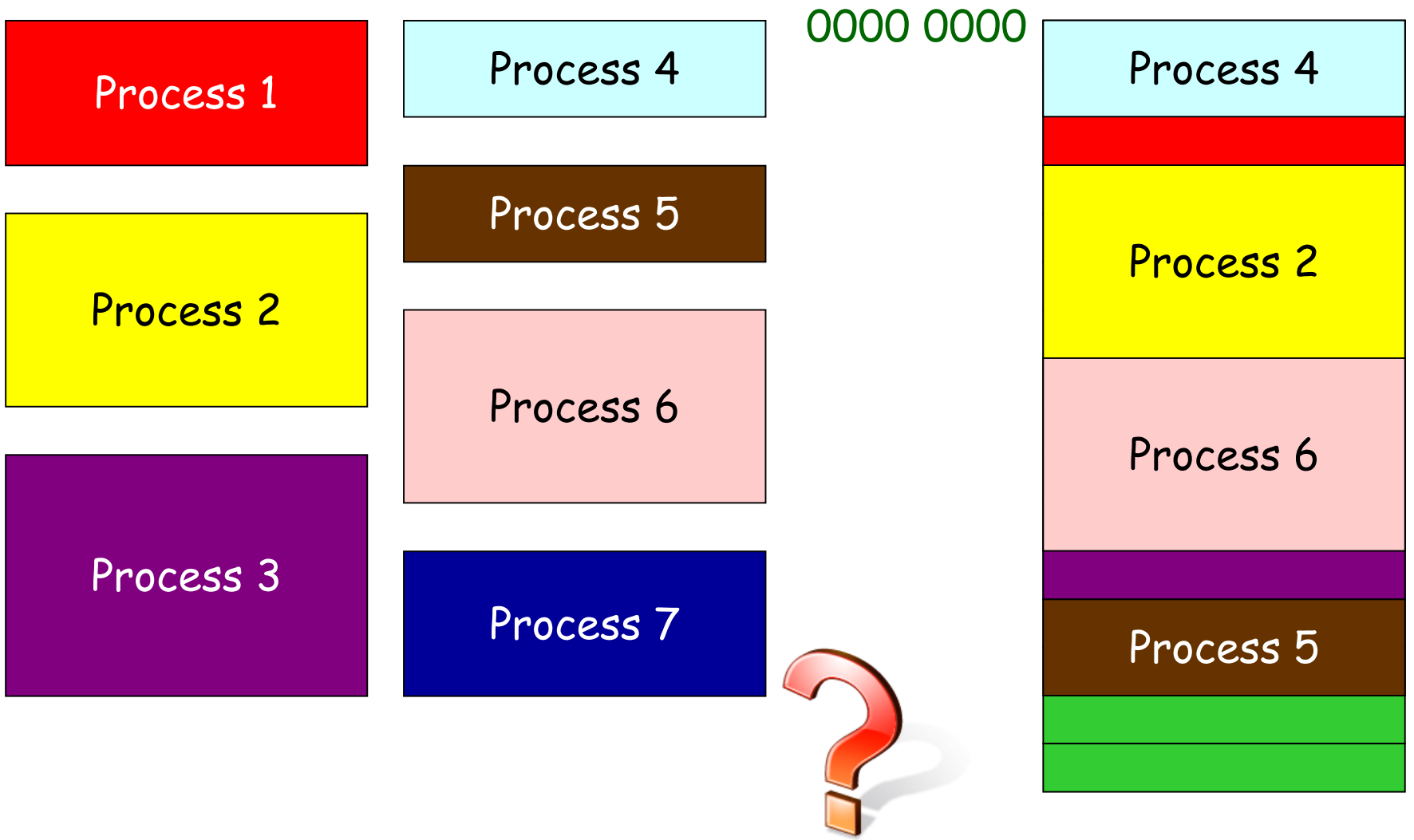
Next-fit



Exercise

- Given memory partitions, which are 100K, 450K, 250K, 300K and 600K
- There are four processes in order: 212K, 417K, 112K, and 426K.
- In order to place processes in the memory, there are three algorithms: the first-fit algorithm, the best-fit algorithm, and the next-fit algorithm. Of the three algorithms, which one makes the best use of memory space?

Fragmentation



Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
  - External fragmentation is the phenomenon in which free storage becomes divided into many small pieces over time.
  - The term "**external**" refers to the fact that the unusable storage is outside the allocated regions.

Fragmentation

- Consider a multiple-partition allocation scheme with a hole of 18,464 bytes. Suppose that the next process requested 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.

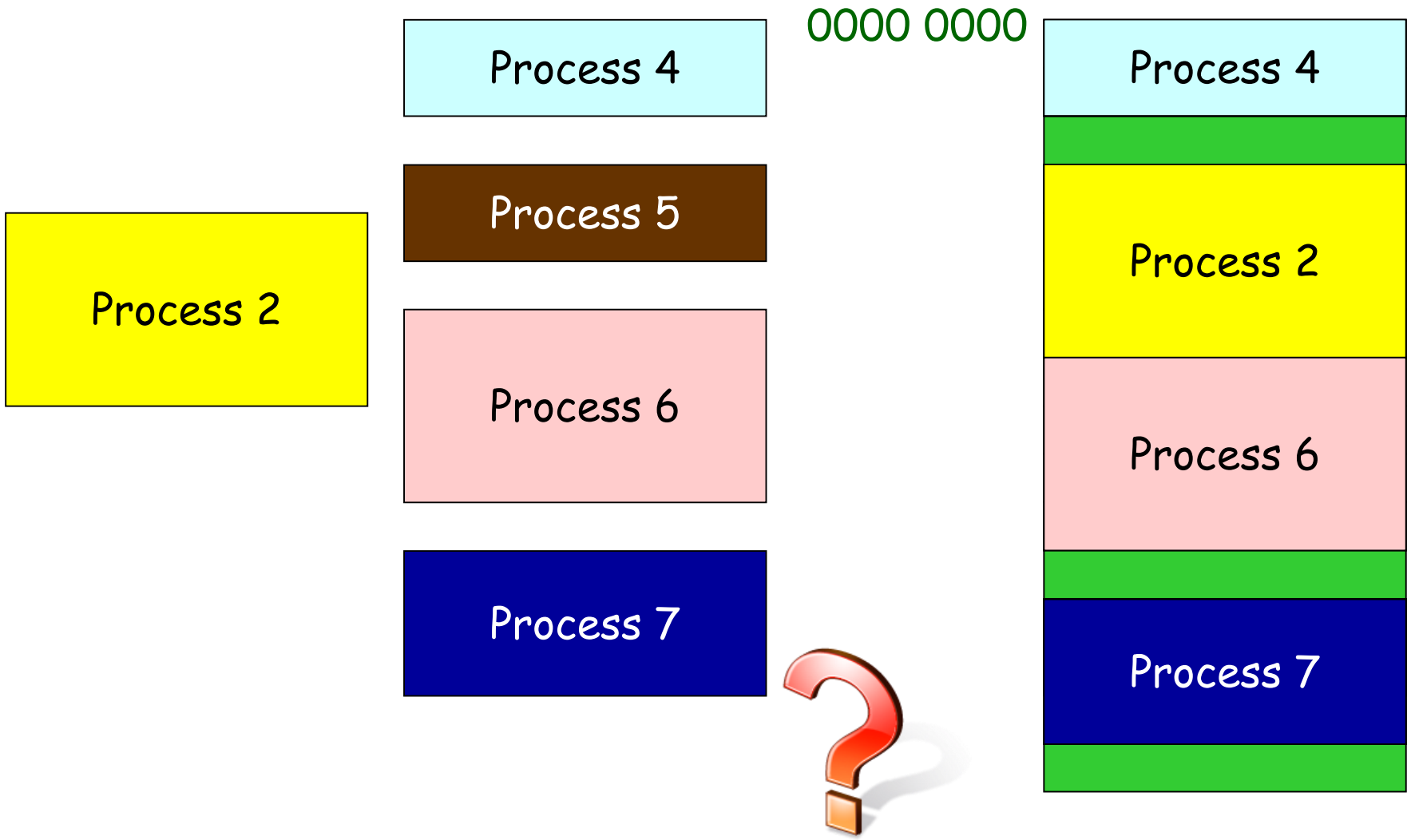
Fragmentation

- 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.
- With this approach, the memory allocated to a process may be slightly larger than the requested memory.
- The difference between these two approaches is **internal fragmentation** - memory that is internal to a partition but is not being used.

Fragmentation

- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory together in one large block
    - However, it is somewhat expensive.
  - Compaction is possible only if relocation is dynamic, and is done at execution time

Compaction


Process 4
Process 5
Process 6
Process 7
Process 2
0000 0000
Process 4
Process 2
Process 6
Process 7

Contiguous Memory Allocation

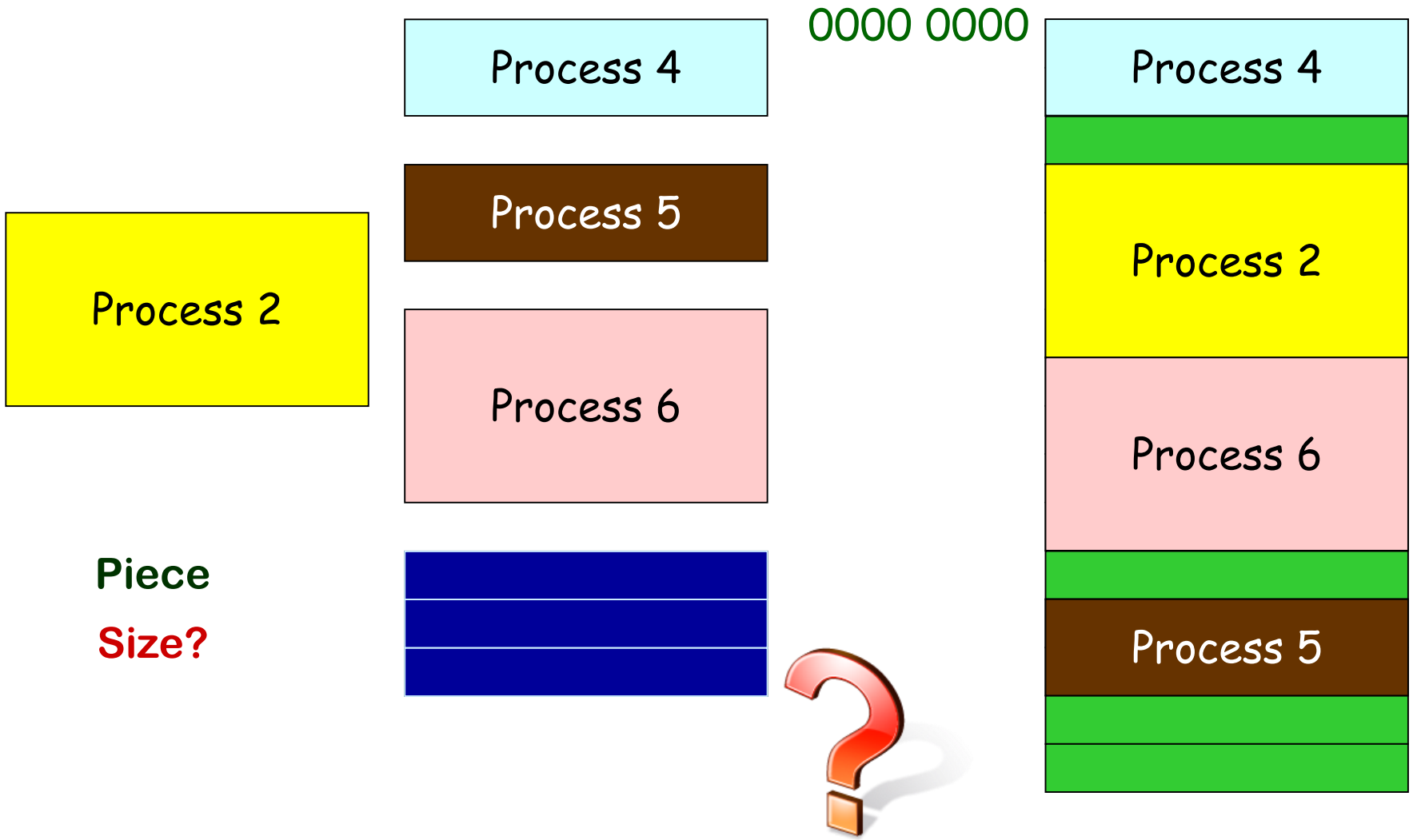
- Base and Limit Pros: Simple, fast
- Fragmentation problem
  - Not every process is in the same size
  - Over time, memory space becomes fragmented
- Missing support for sparse address space
  - Would like to have multiple chunks/program
    - e.g.: Code, Data, Stack
- Hard to do inter-process sharing
  - Want to share code segments when possible
  - Want to share memory between processes

Contiguous Memory Allocation

- Is there a scheme without External Fragmentation?
  - Of course!

Paging

Paging

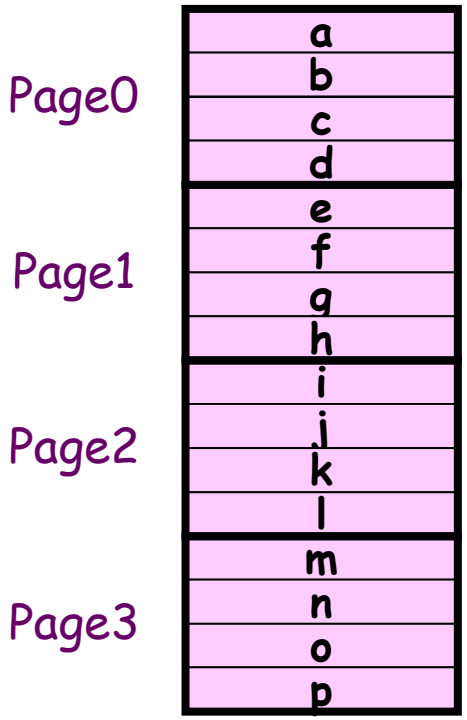

Process 4
Process 5
Process 2
Process 6
Piece
Size?
0000 0000
Process 4
Process 2
Process 6
Process 5

Paging

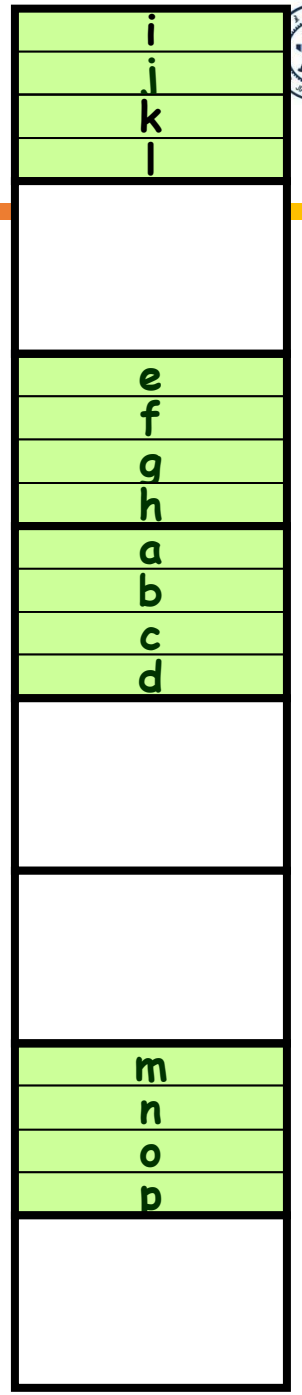
- Logical address space of a process can be **noncontiguous**; process is allocated physical memory whenever the latter is available
- Divide physical memory into fixed-sized blocks called **frames**
  - Size is power of 2
  - Typically have small pages (1K-16K)
- Divide logical memory into blocks of same size called **pages**

How to allocate?

- 32-byte memory and 4-byte pages



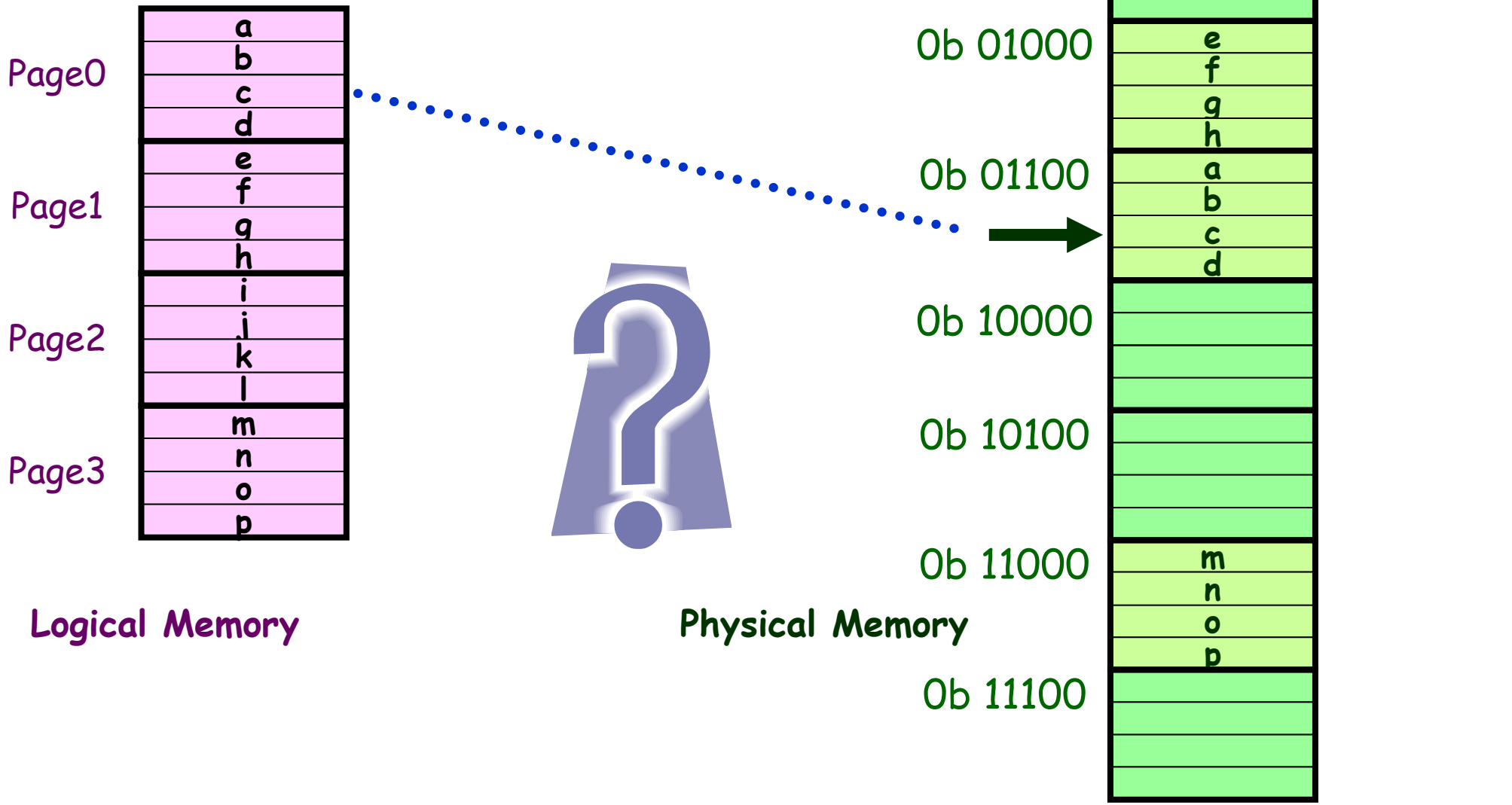
Logical Memory



Physical Memory

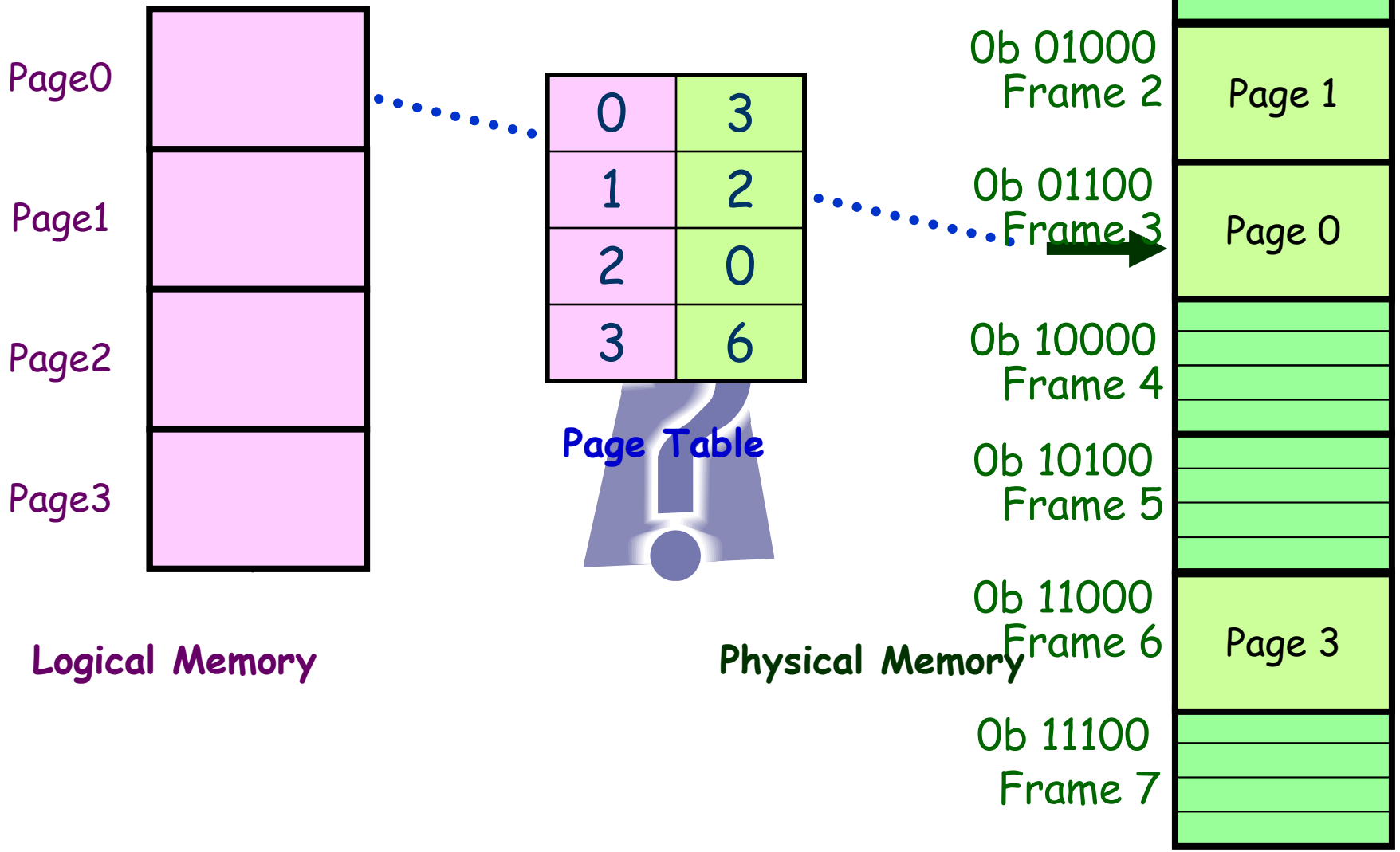
How to allocate?

- 32-byte memory and 4-byte pages



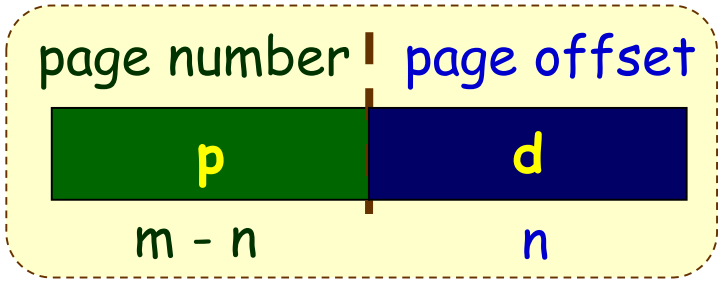
How to allocate?

- 32-byte memory and 4-byte pages



Address Translation Scheme

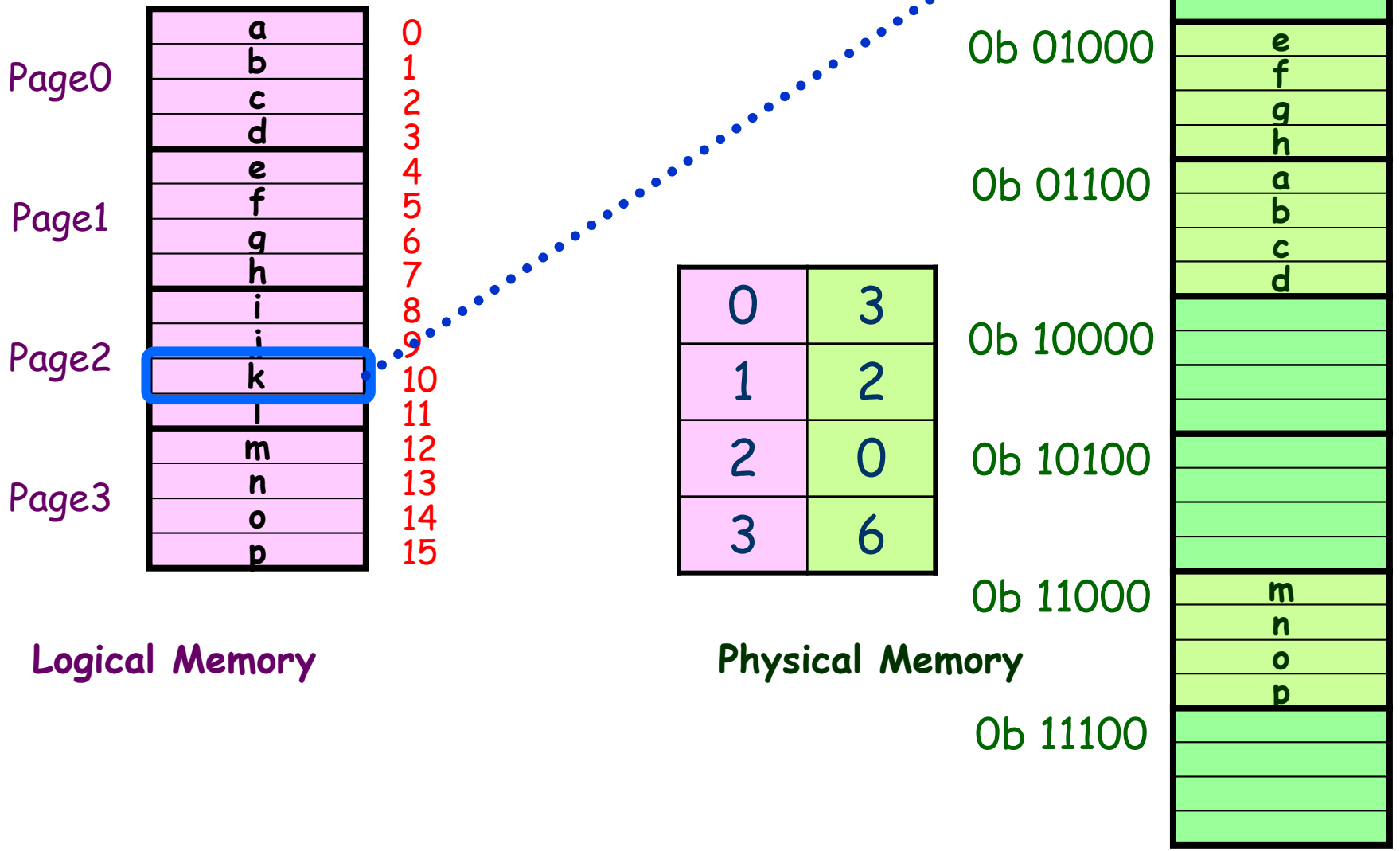
- Address generated by CPU is divided into:
  - Page number (p) – used as an index into a page table which contains base address of each page in physical memory
  - Page offset (d) – combined with base address to define the physical memory address that is sent to the memory unit



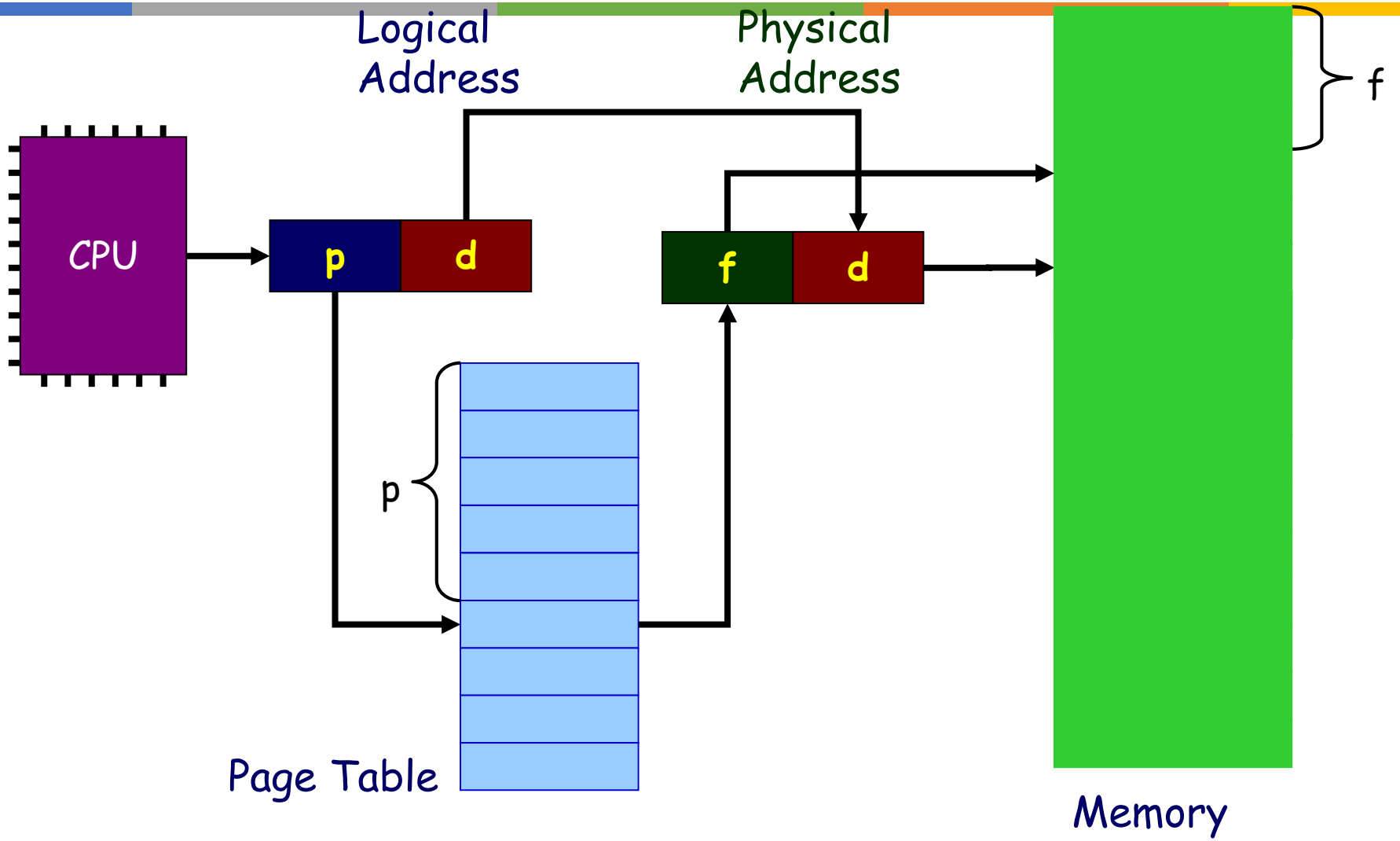
- For given logical address space 2^m and page size 2^n

How to allocate?

- 32-byte memory and 4-byte pages



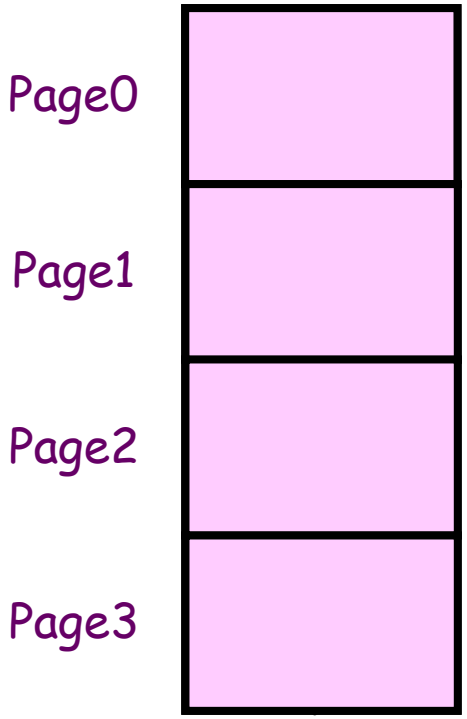
Paging Hardware



Paging

- 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
- No external fragmentation
- May has internal fragmentation

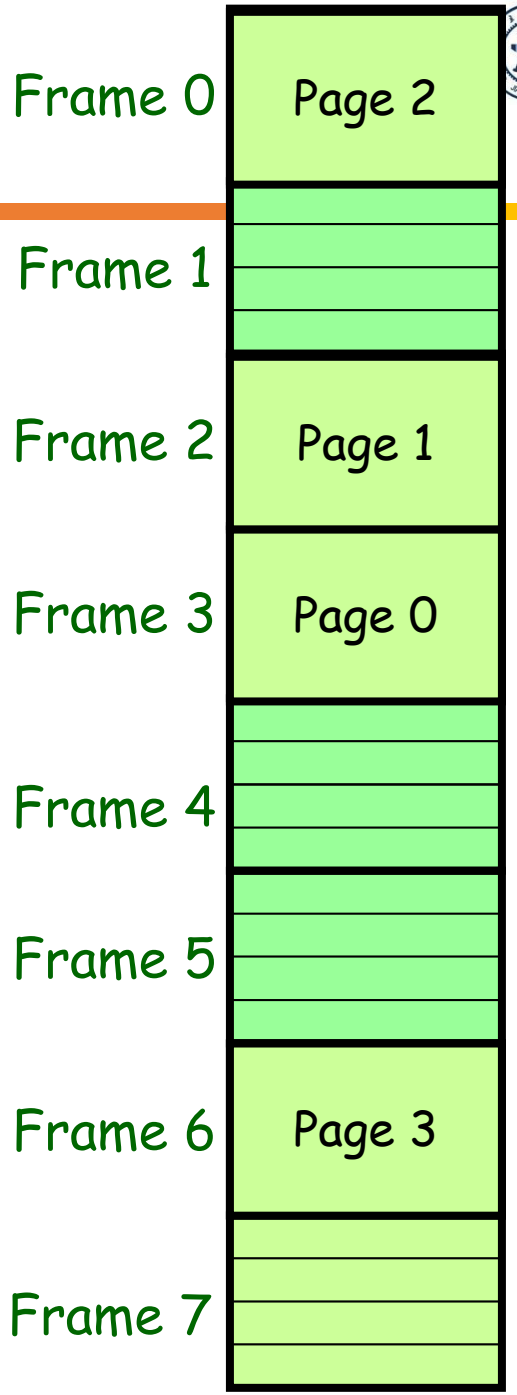
Free Frames



Logical Memory

0	3
1	2
2	0
3	6

Page Table



Physical Memory

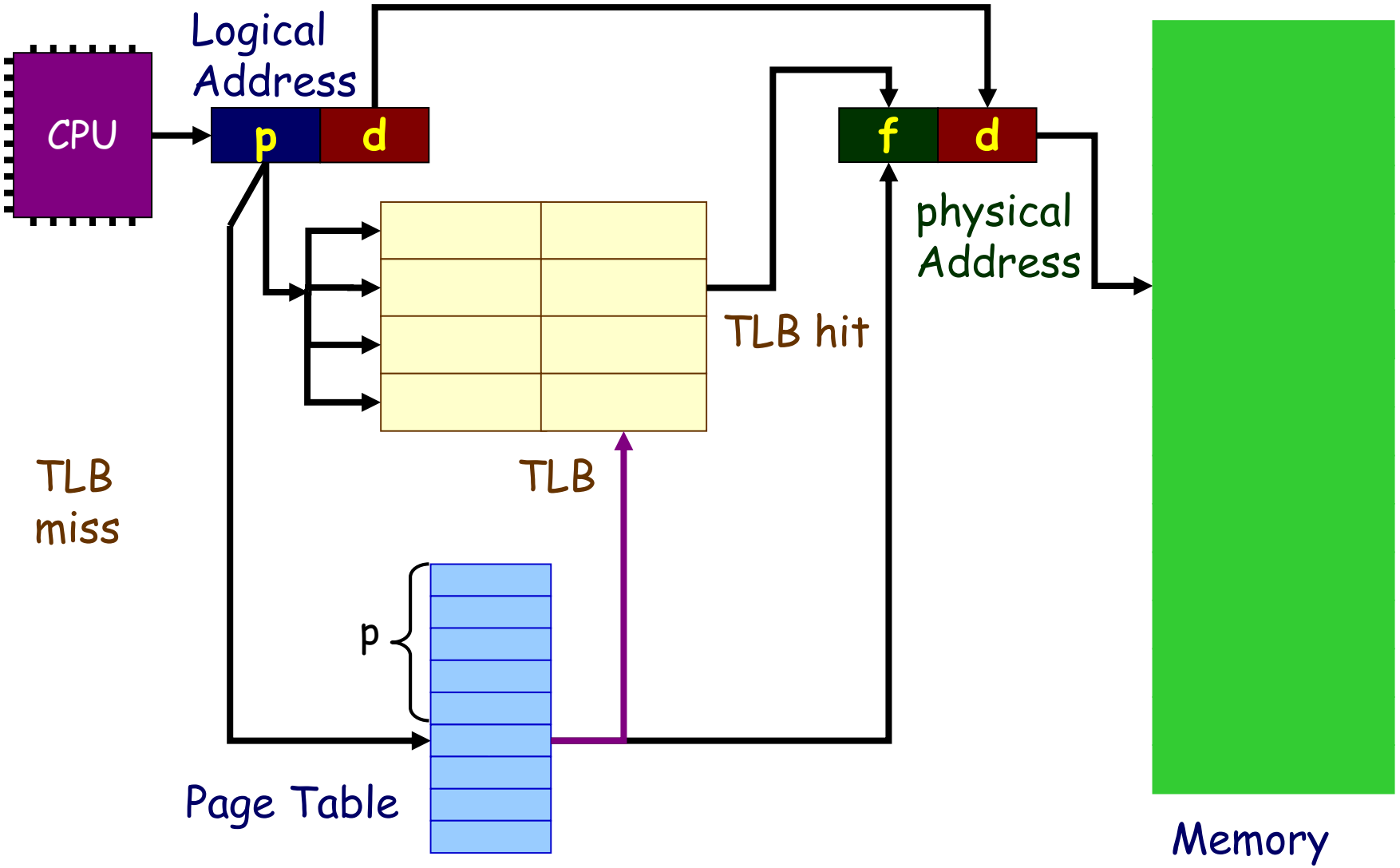
Implementation of Page Table

- The page table is contiguous in main memory
- Page-table base register (PTBR) points to the page table
- Page-table length register (PRLR) indicates size of the page table

Implementation of 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)**

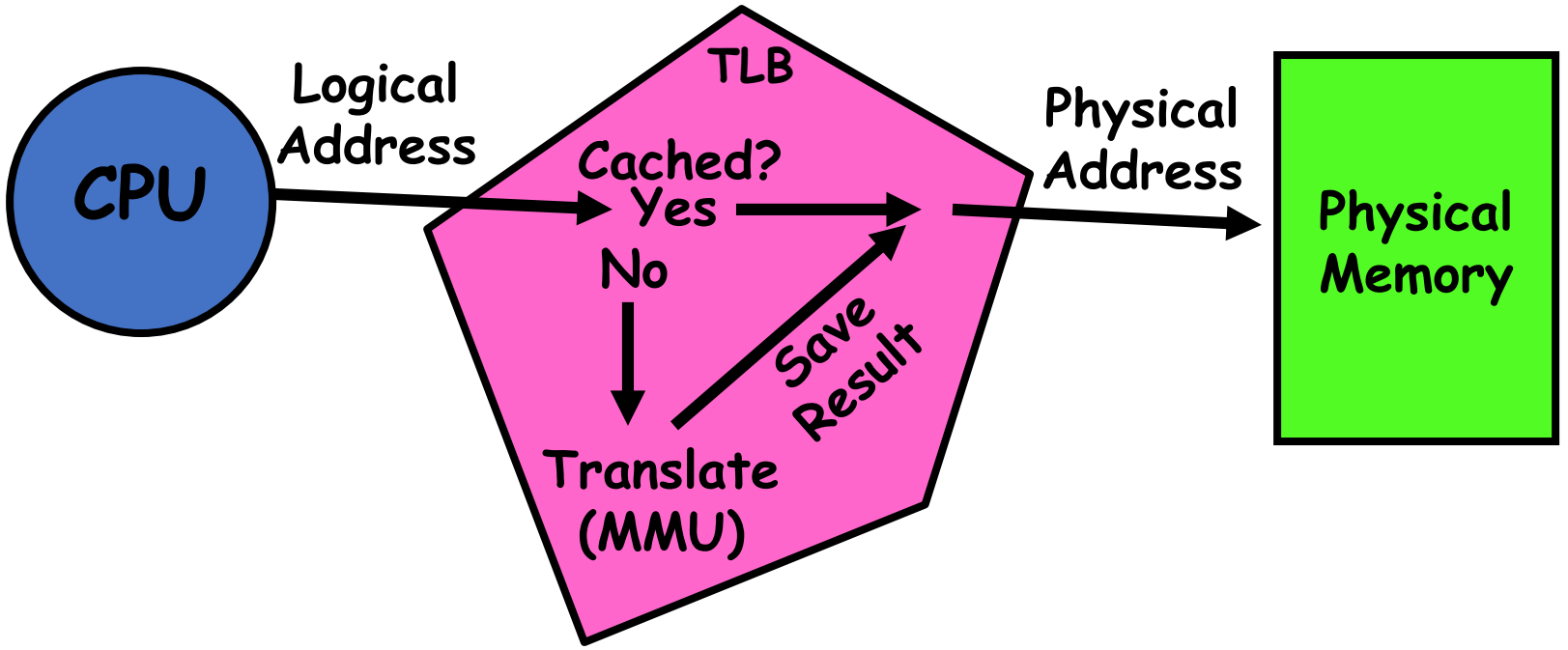
Paging Hardware With TLB



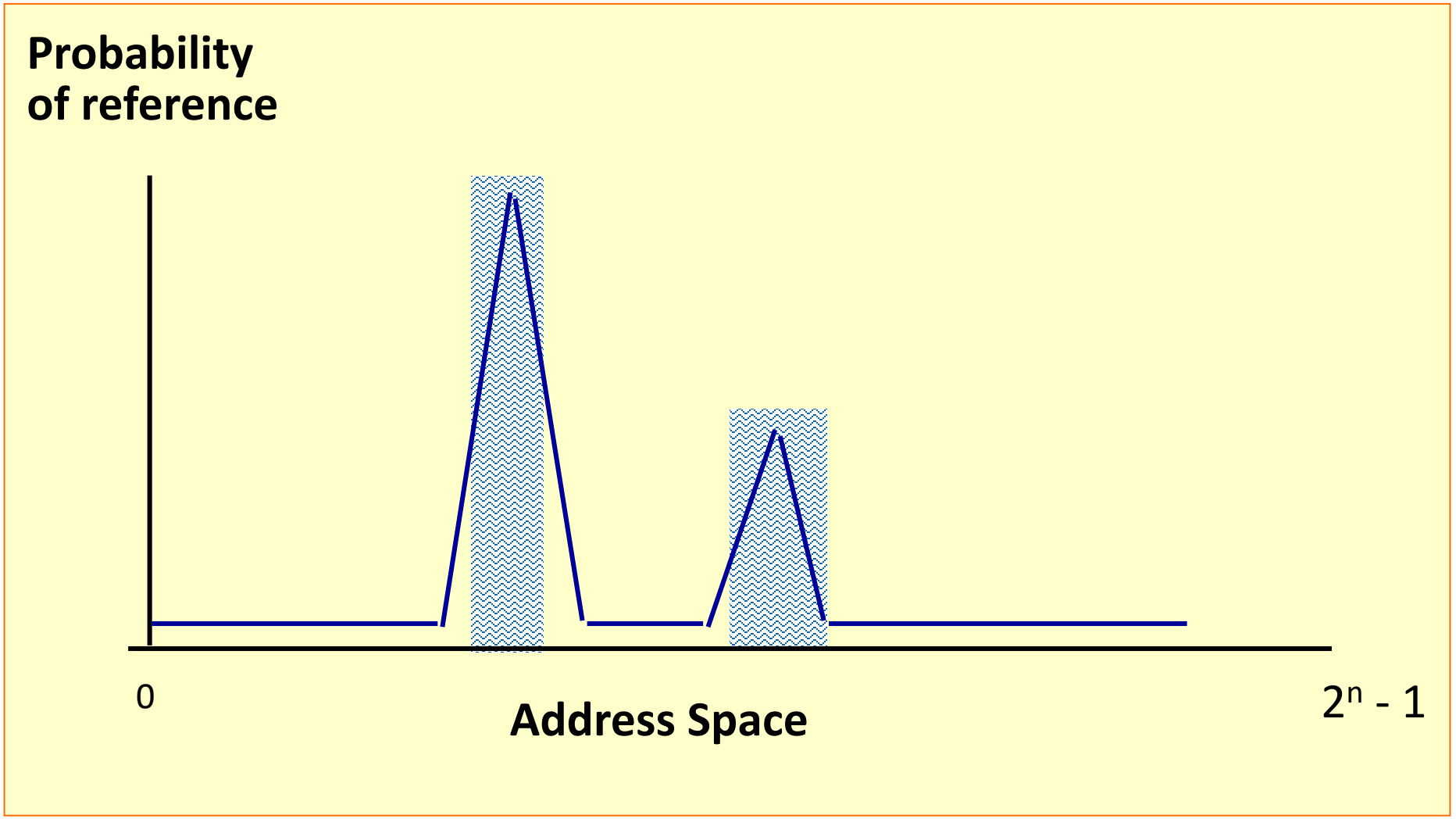
Paging Hardware With TLB

- If the page number is not in the TLB (known as a TLB miss), a memory reference to the page table must be made. When the frame number is obtained, we can use it to access memory.
- In addition, we add the page number and frame number to the TLB, so that they will be found quickly on the next reference.
- If the TLB is already full of entries, the operating system must select one for replacement.

Paging Hardware With TLB



Why Does TLB Help? Locality!



Effective Access Time

- Associative Lookup = ε time unit
- Assume memory cycle time is 1 time unit
- **Hit ratio** – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
  - Hit ratio = α
- Effective Access Time (EAT)

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

Effective Access Time

ε=0.1

EAT = $2 + \varepsilon - \alpha$

α	EAT
5%	2.05
10%	2
50%	1.6
80%	1.3
?	1.12

Paging Hardware With TLB

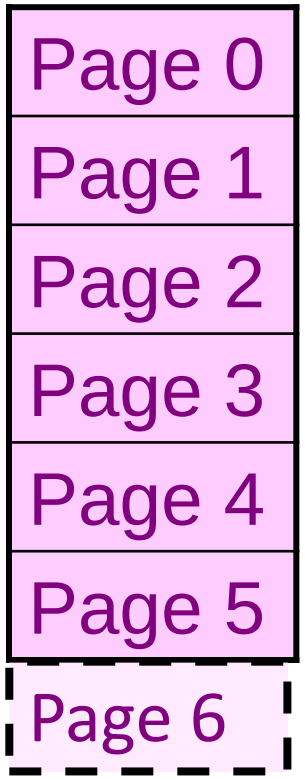
- Needs to be really fast
  - Critical path of memory access
- Needs to have very few conflicts!
  - With TLB, the Miss Time extremely high!
- How big does TLB actually have to be?
  - Usually small: 128-512 entries

Memory Protection

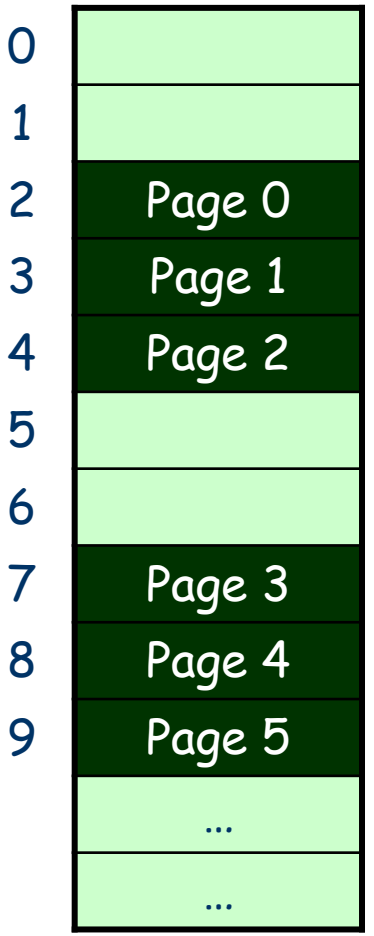
- User cannot modify the page table mapping
- Memory protection implemented by associating protection bit with each frame
- Valid-invalid bit attached to each entry in the page table:
  - "valid" indicates that the associated page is in the process' logical address space, and is thus a legal page
  - "invalid" indicates that the page is not in the process' logical address space

Paging

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



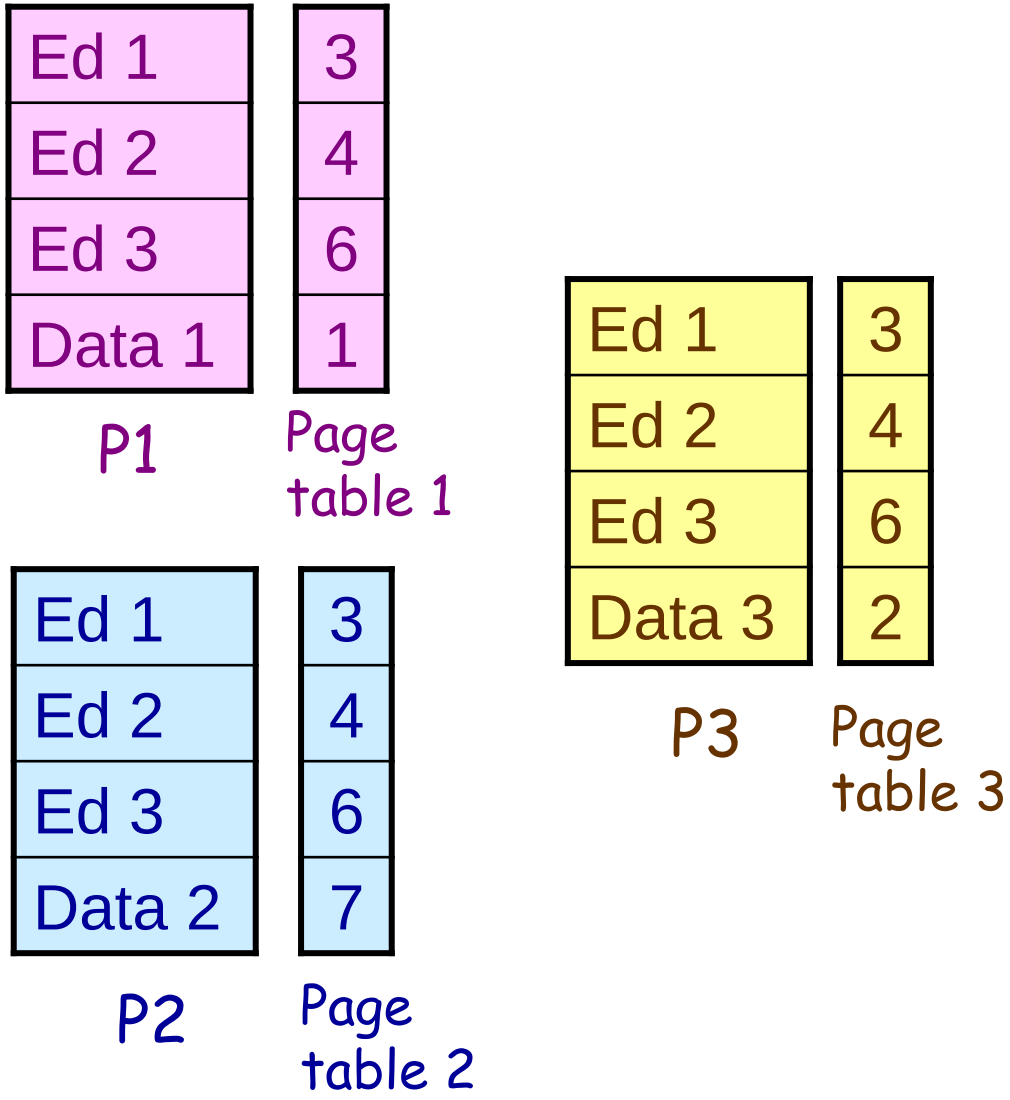
	Frame number	Valid-invalid bit
0	2	v
1	3	v
2	4	v
3	7	v
4	8	v
5	9	v
6	0	i
7	0	i



Shared Pages

- Shared code
  - One copy of read-only code shared among processes (i.e., text editors).
- 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



Page Table Structure

Structure of the Page Table

- 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
- How many pages?
- What is the size of the page table, if each item of the page table takes 4B?

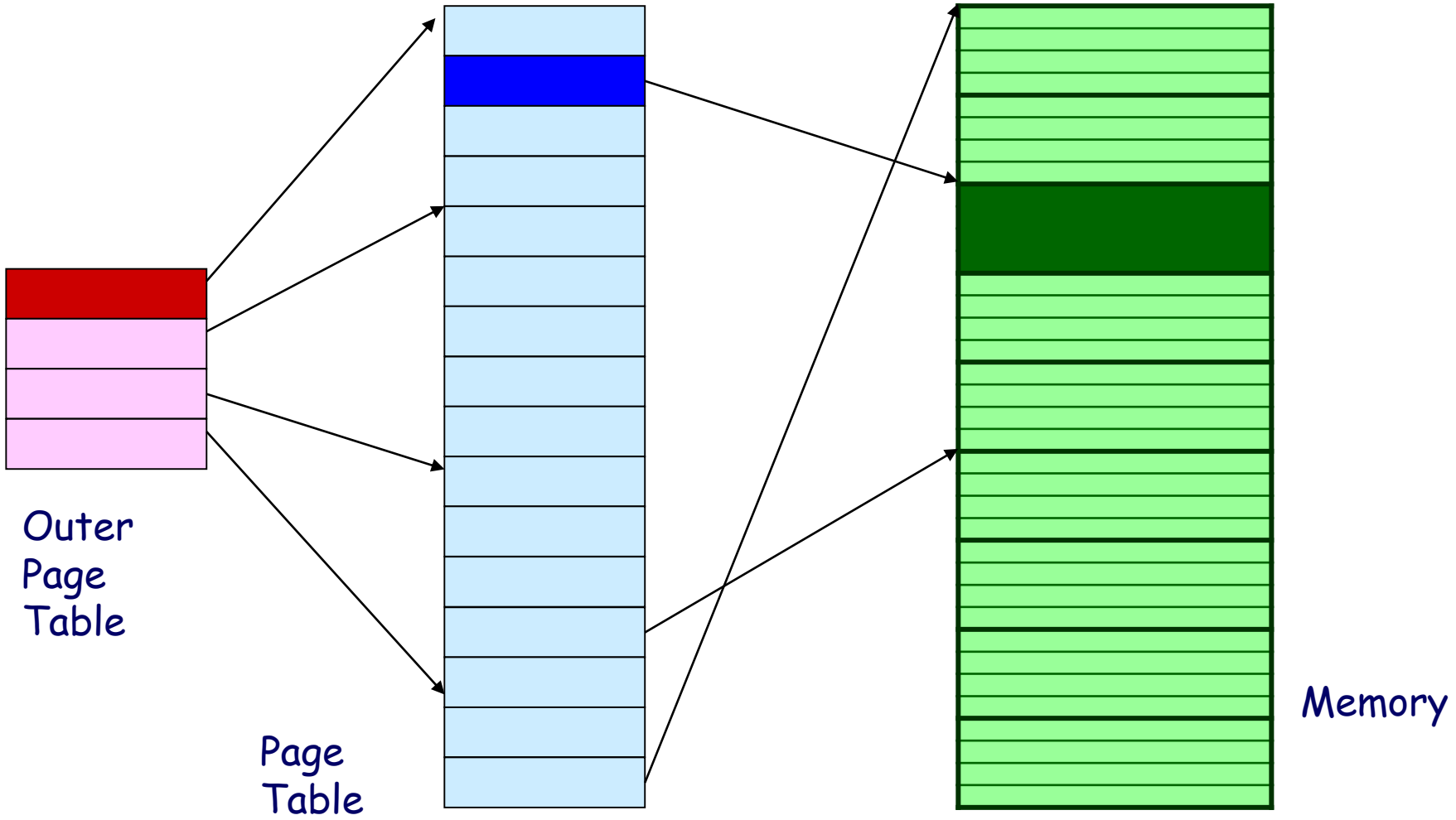
Structure of the Page Table

- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

Hierarchical Page Tables

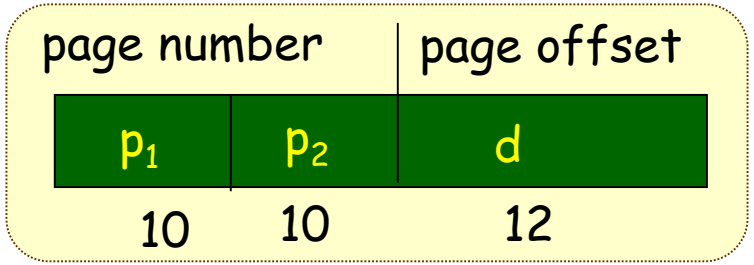
- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table

Two-Level Page-Table Scheme



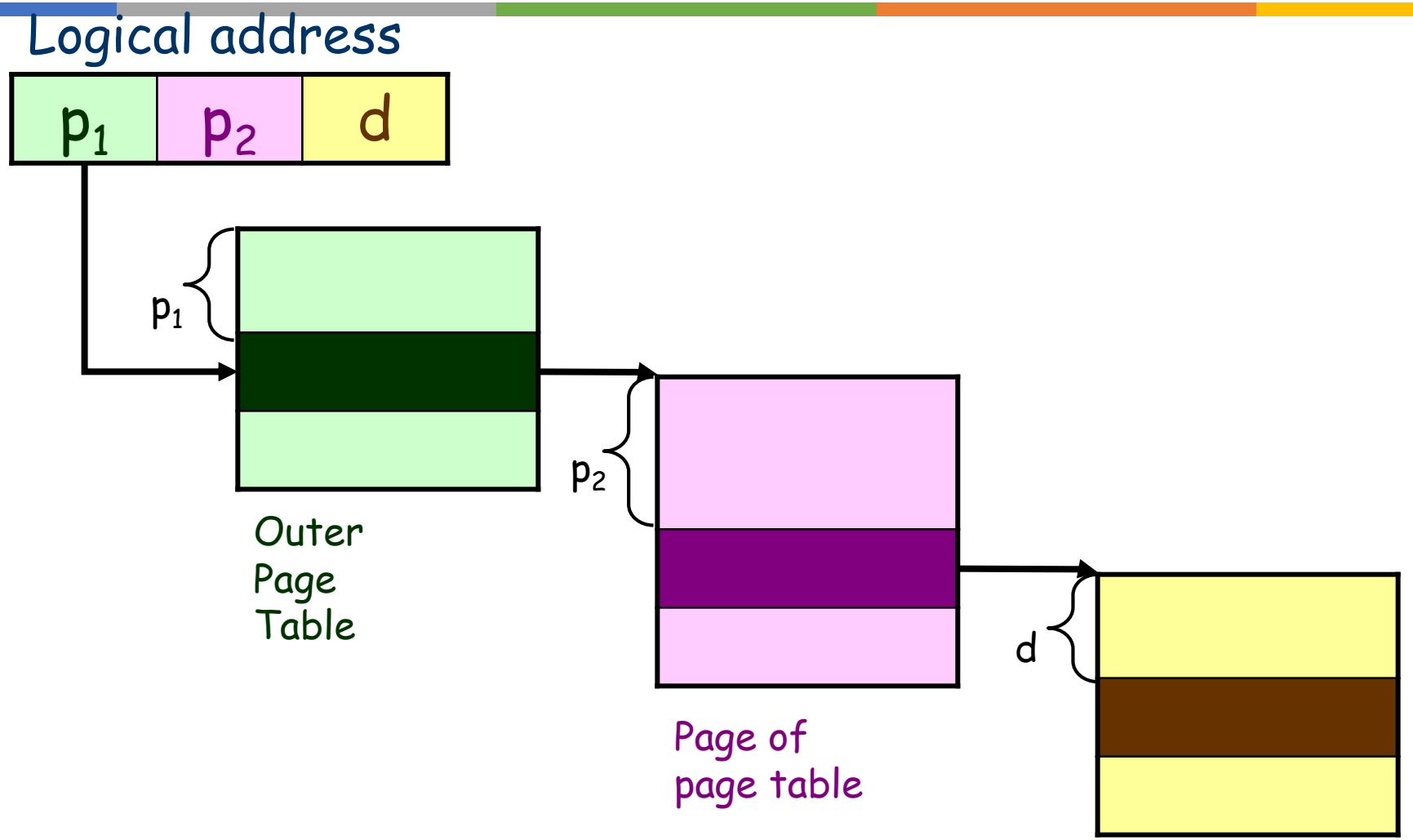
Two-Level Paging Example

- 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
- Since the page table is paged, the page number is further divided into:
  - a 10-bit page number
  - a 10-bit page offset
- Thus, a logical address is as follows:



where p_1 is an index into the outer page table, and $\boldsymbol{p_2}$ is the displacement within the page of the outer page table

Address-Translation Scheme



Three-level Paging Scheme

outer page	inner page	offset
p_1	p_2	d
42	10	12

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

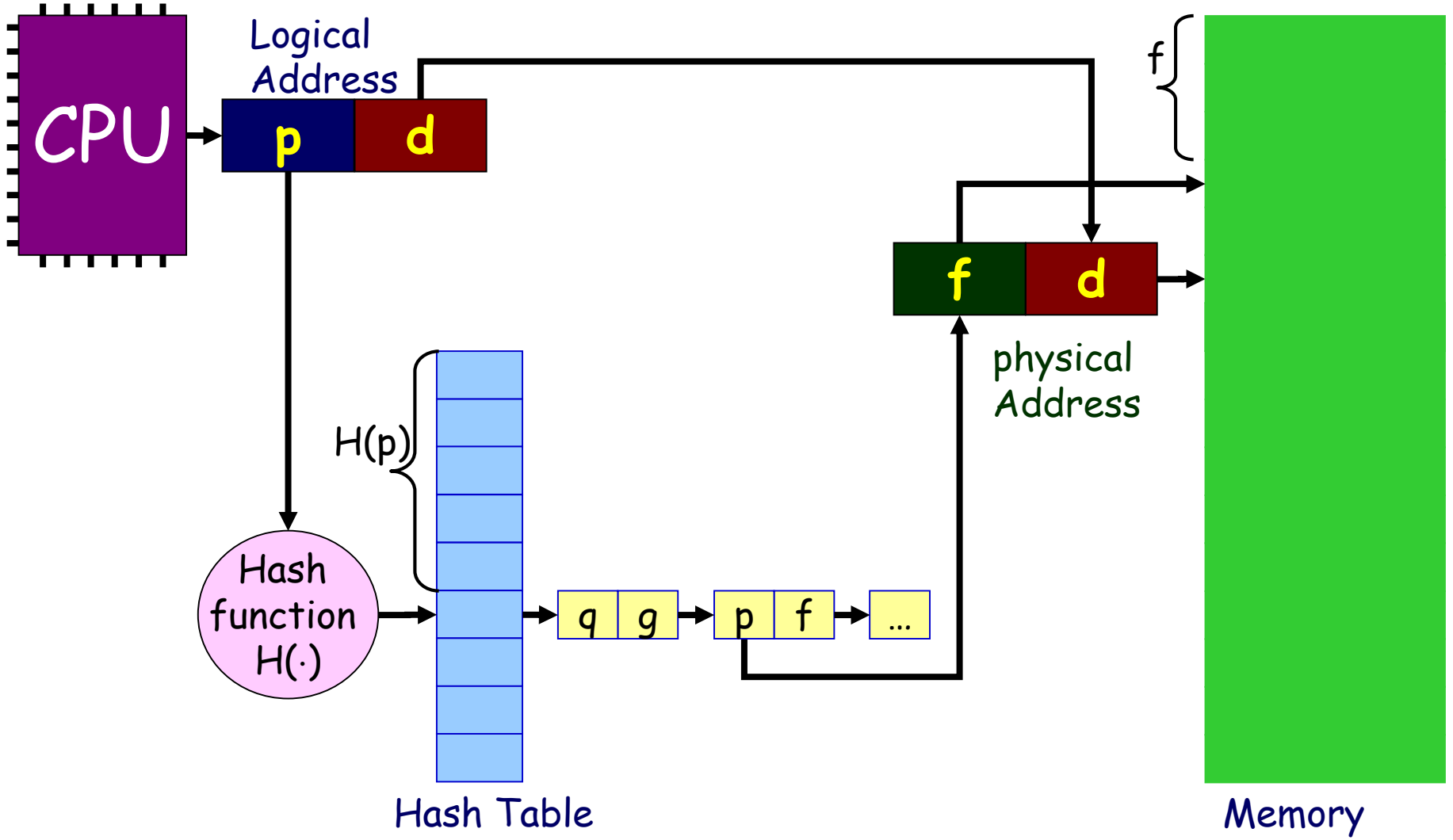
Paging Analysis

- Pros:
  - Easy memory allocation
  - Easy sharing
- Cons:
  - Page tables need to be contiguous
  - Two (or more, if >2 levels) lookups per reference

Hashed Page Tables

- In common address spaces > 32 bits
- The logical page number is hashed into a page table
  - This page table contains a chain of elements hashing to the same location
- Logical page numbers are compared in this chain searching for a match
  - If a match is found, the corresponding physical frame is extracted

Hashed Page Tables



Paging

- Size of page table is at least as large as amount of logical memory allocated to processes
- Physical memory may be much less
  - Much of process space may be out on disk or not in use

Inverted Page Table

P1

0	3	v
1	0	v
2	7	v
3	0	i

P2

0	4	v
1	1	v
2	0	i
3	0	i

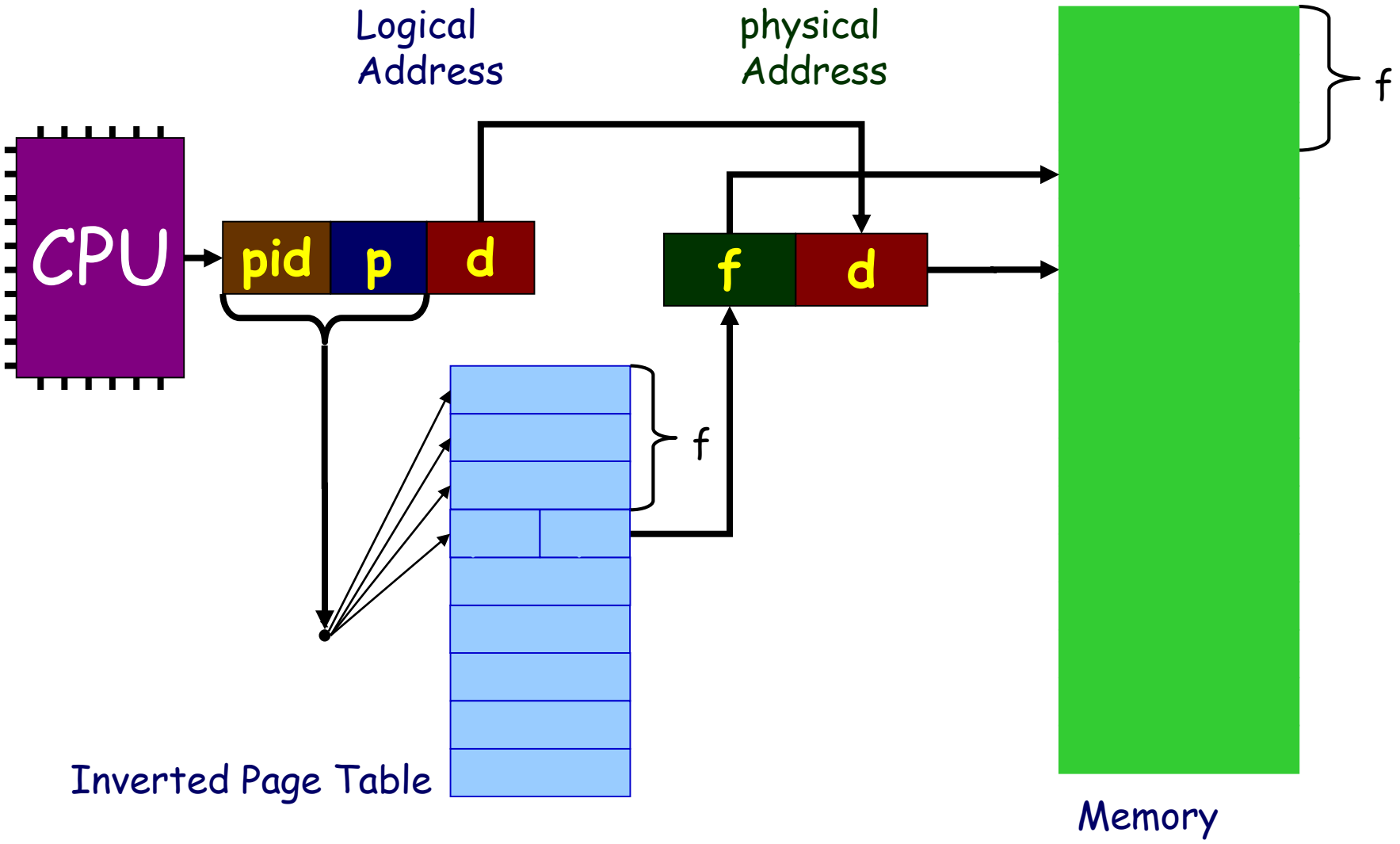
P3

0	5	v
1	2	v
2	6	v
3	0	i
4	0	i

0	P1	Page1
1	P2	Page1
2	P3	Page1
3	P1	Page0
4	P2	Page0
5	P3	Page0
6	P3	Page2
7	P1	Page2

Inverted Page Table

Inverted Page Table



Inverted Page Table

- One entry for each real page of memory
- Entry consists of the logical 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

Inverted Page Table

- Systems that use inverted page tables have difficulty implementing shared memory.

Segmentation

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,

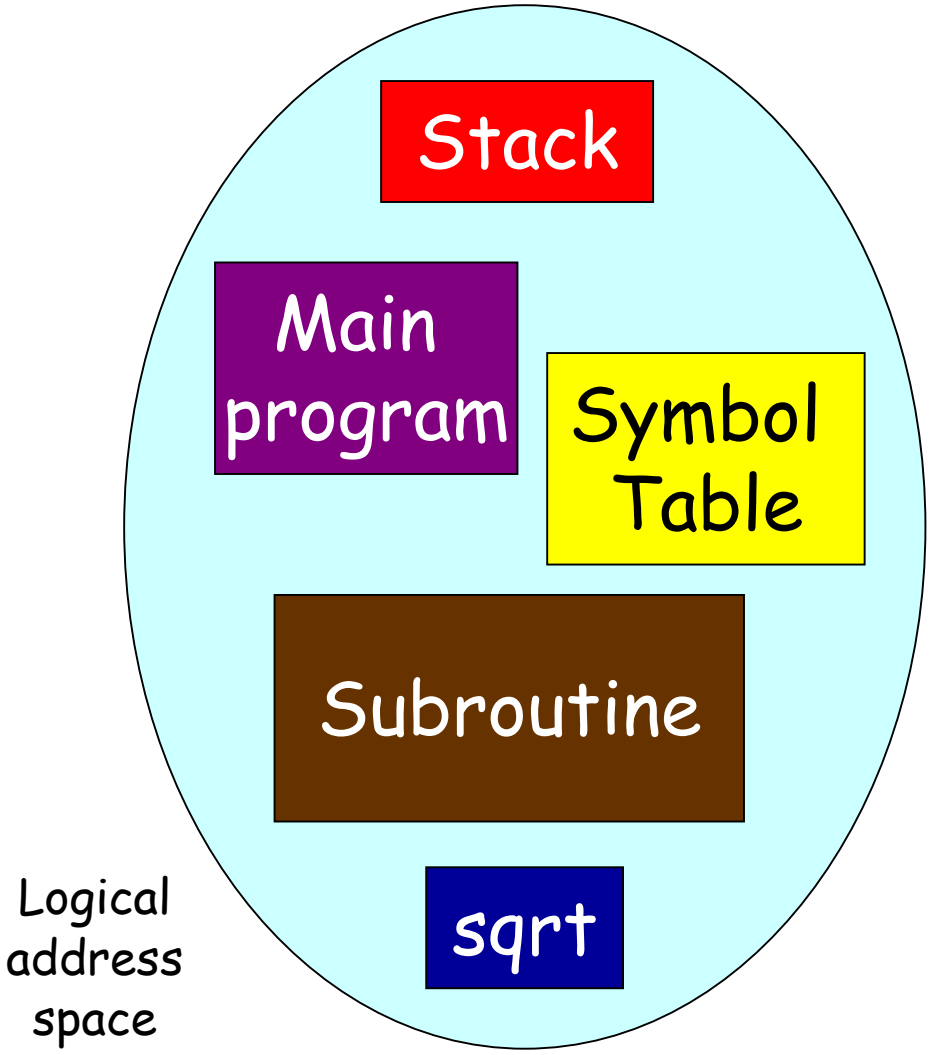
  function,

  local variables, global variables,

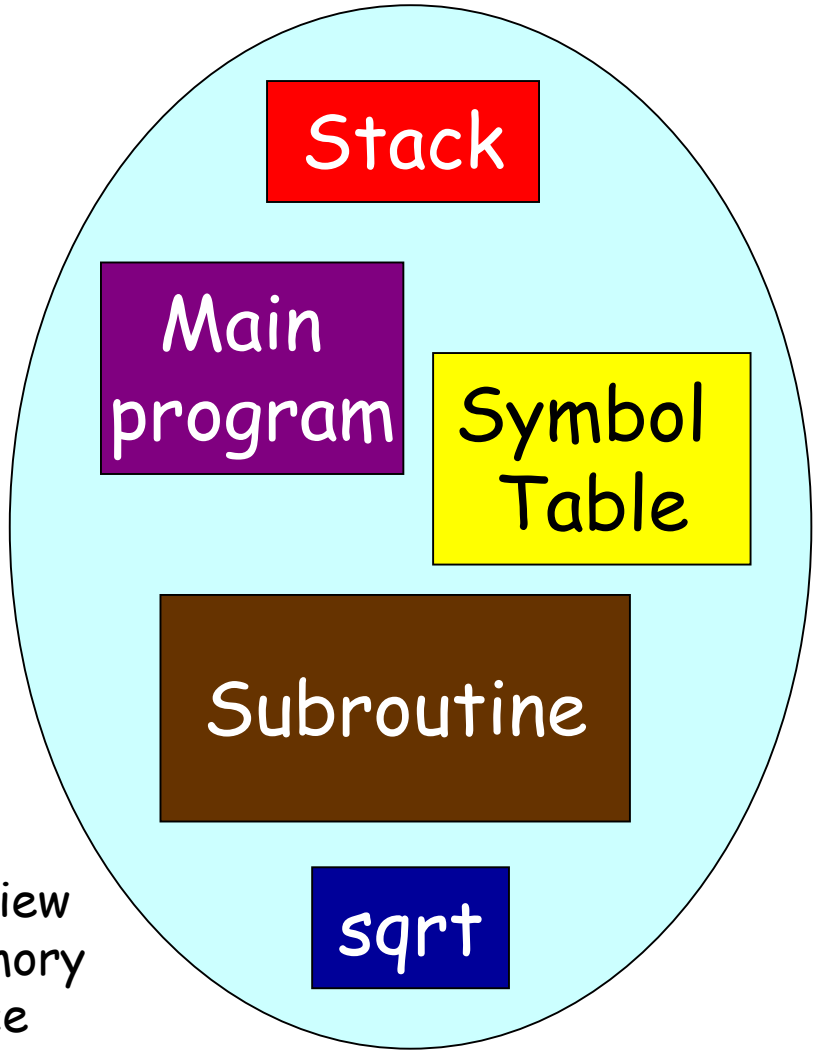
  stack,

  ……

User's View of a Program

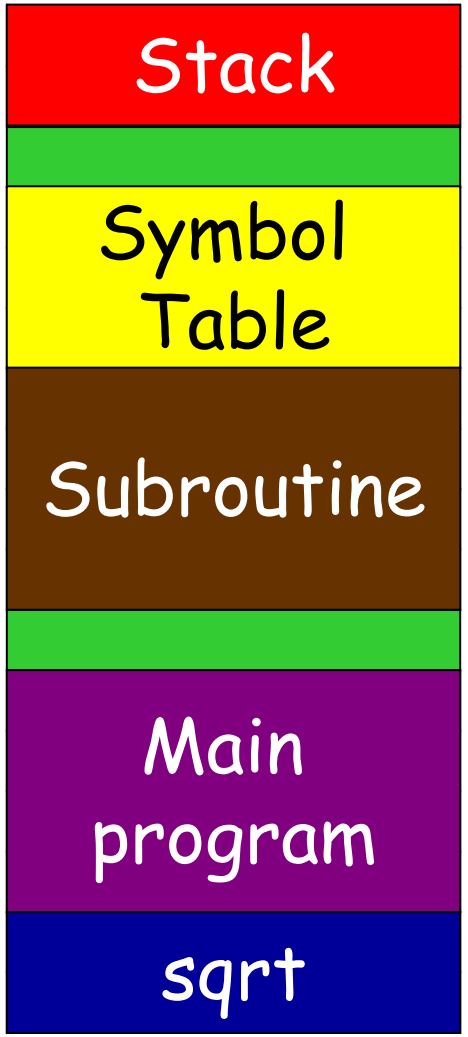


Logical View of Segmentation



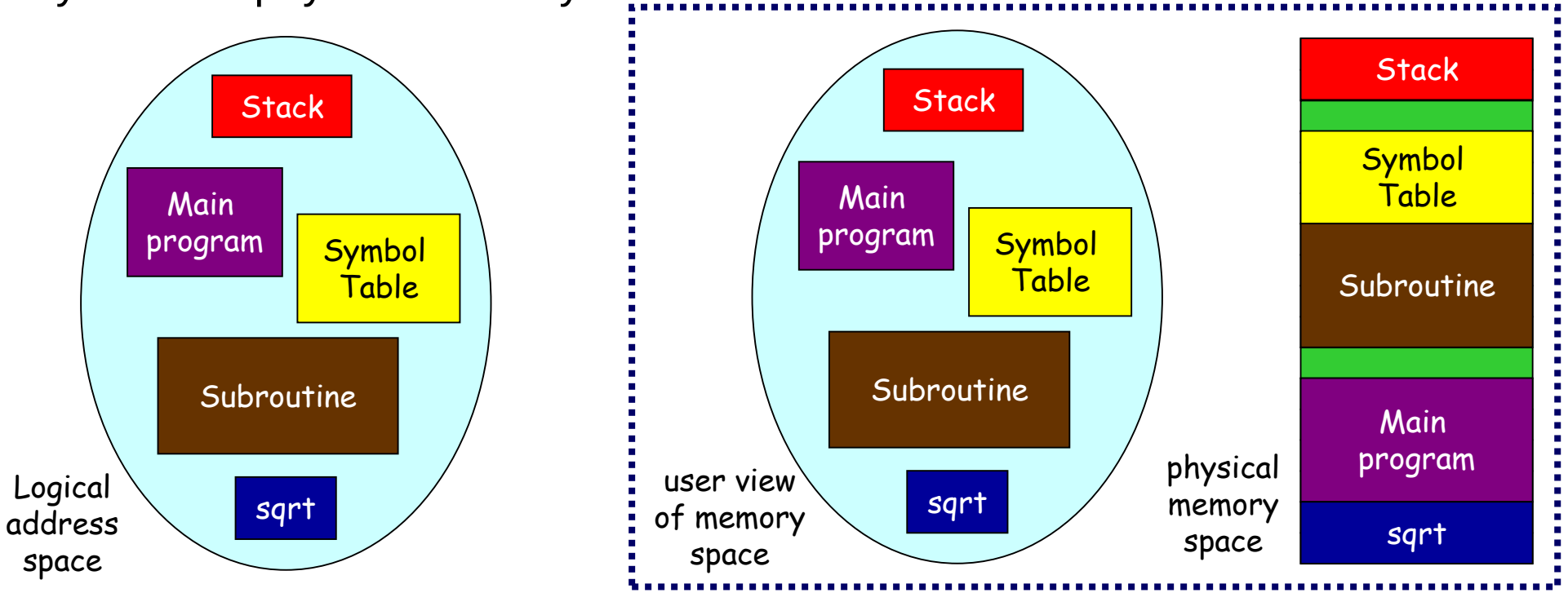
user view of memory space

physical memory space



More Flexible Segmentation

- **Logical View: multiple separate segments**
  - Typical: Code, Data, Stack
  - Others: memory sharing, etc.
- **Each segment is given region of contiguous memory**
  - Has a base and limit
  - Can reside anywhere in physical memory



Segmentation Architecture

- Logical address consists of a two tuple :

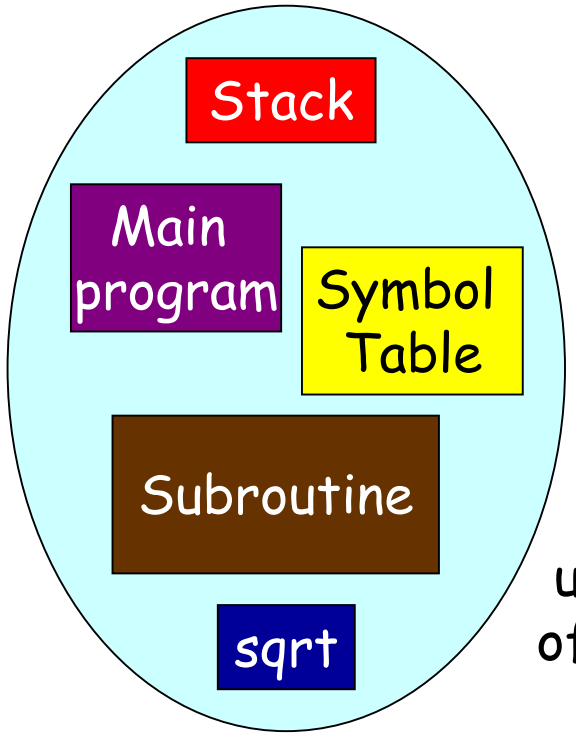
  <segment-number, offset>

- **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

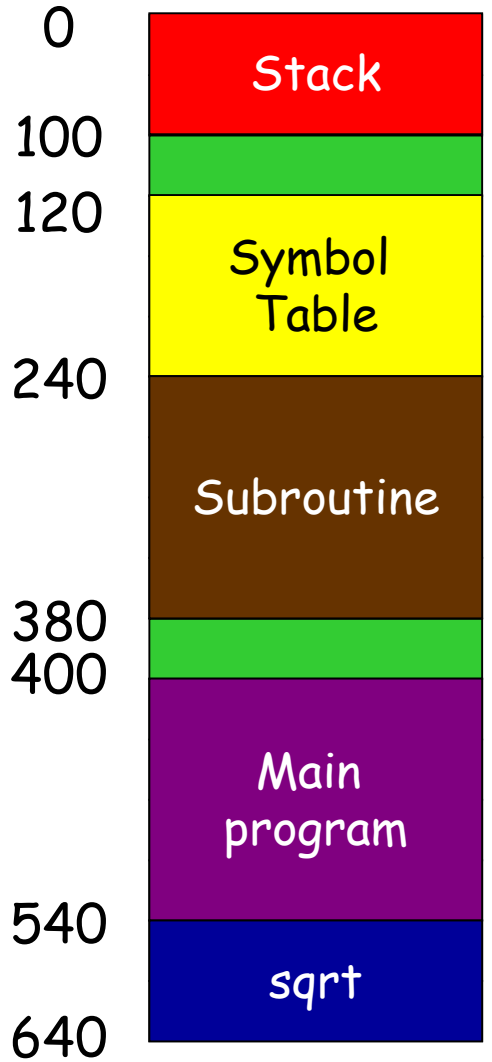
Segmentation Architecture

	limit	base
0	100	0
1	120	120
2	140	240
3	140	400
4	100	540

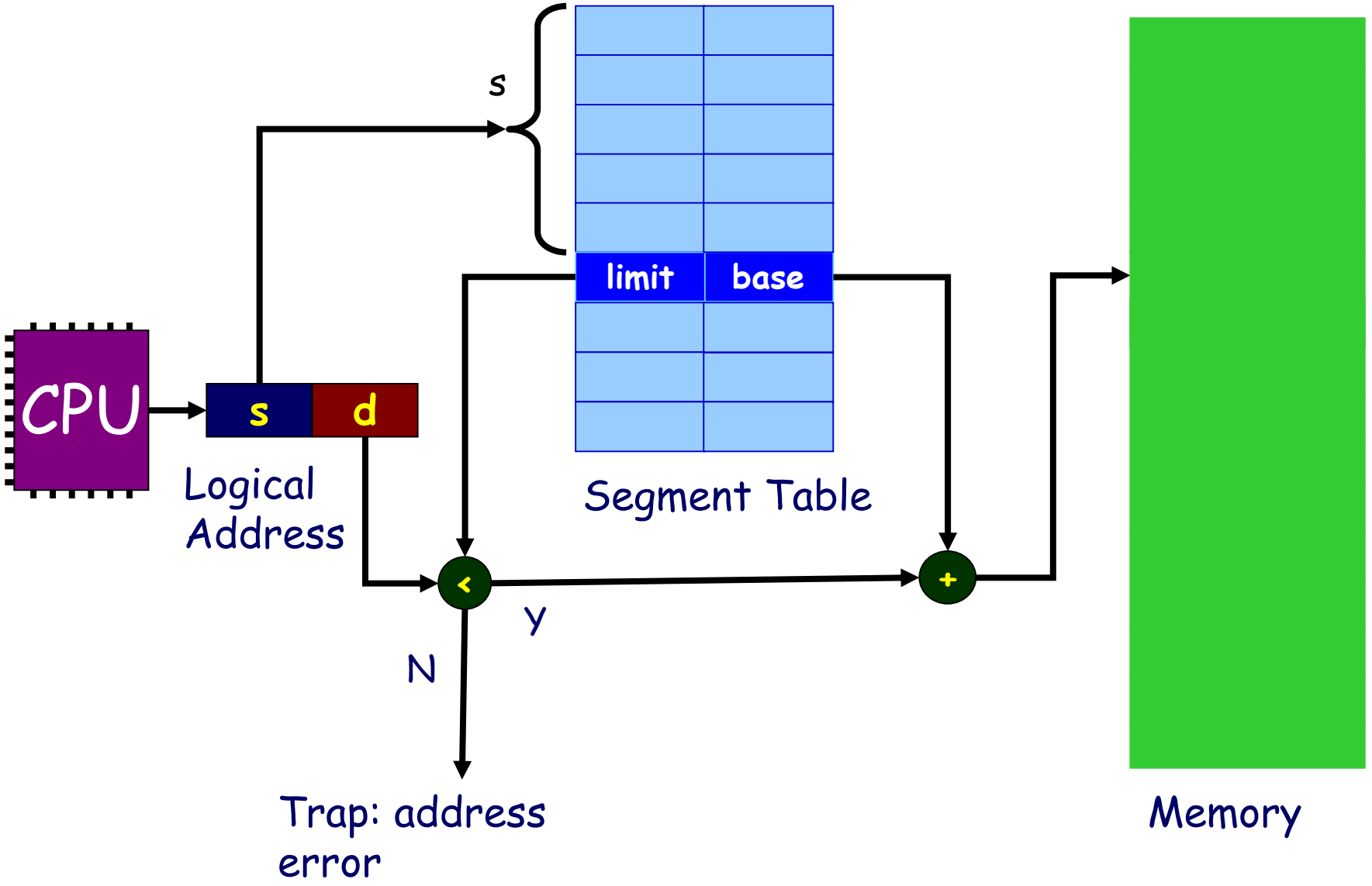


user view of memory space

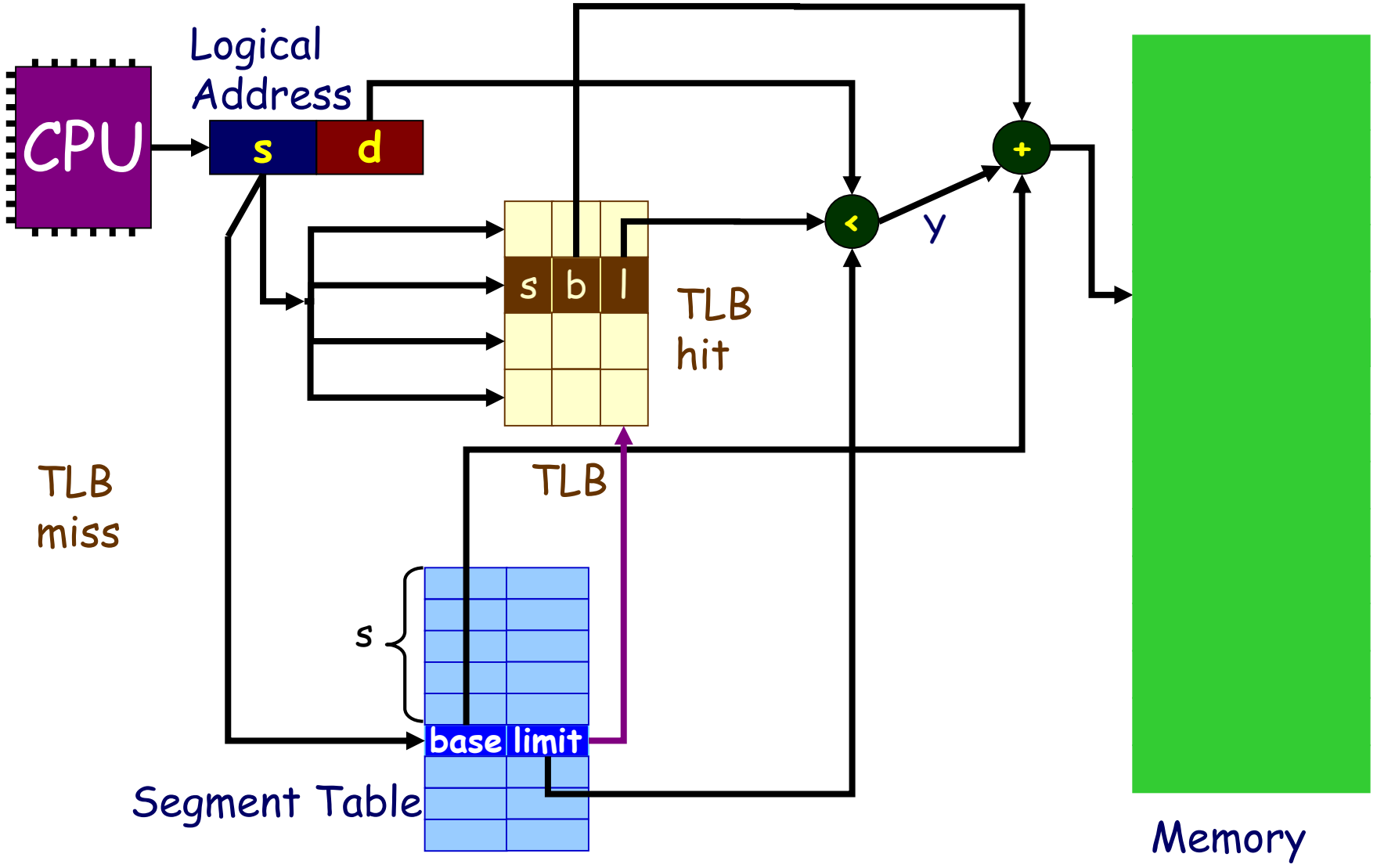
physical memory space



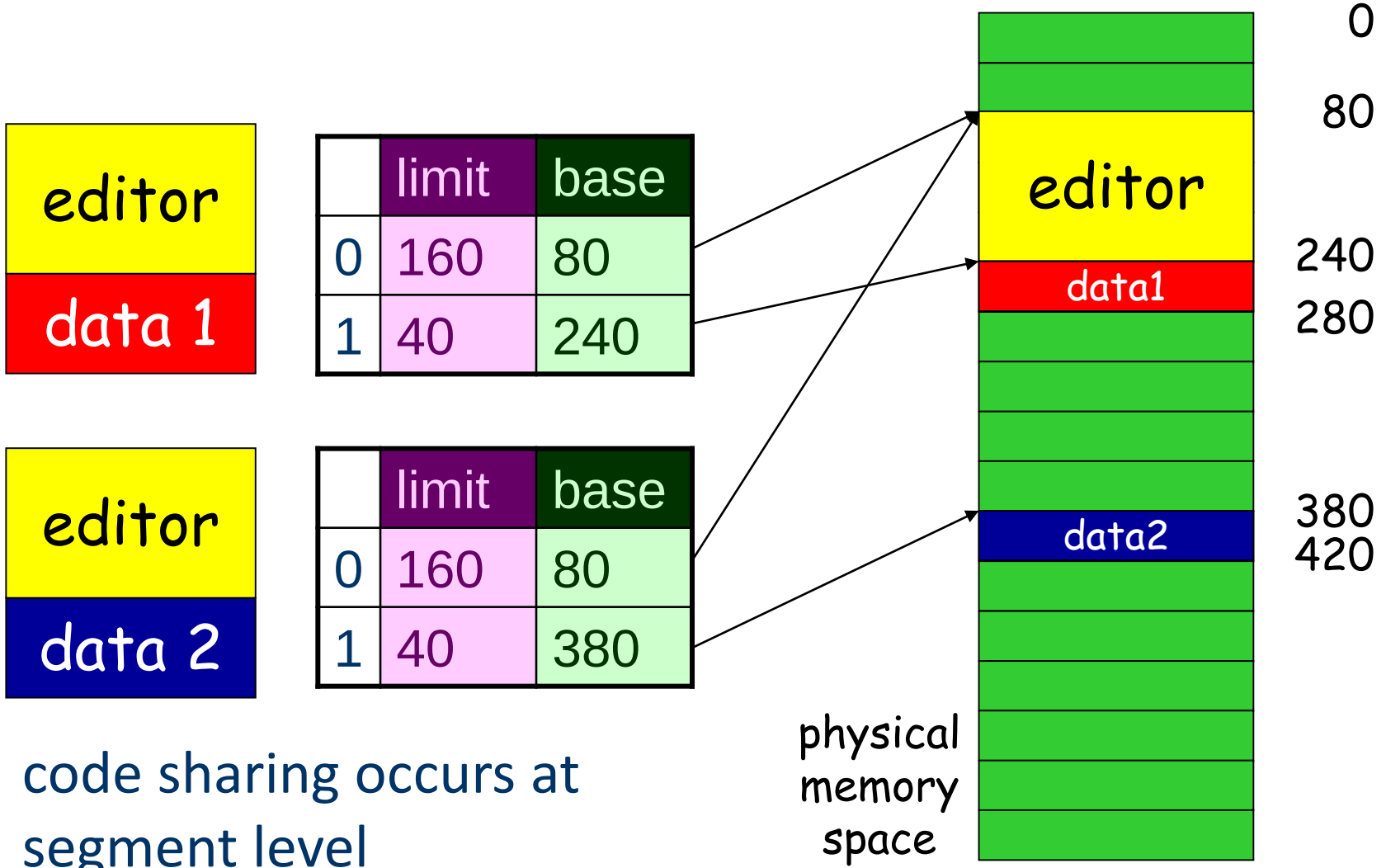
Segmentation Hardware



Segmentation Hardware With TLB



Shared Segments Example



code sharing occurs at segment level

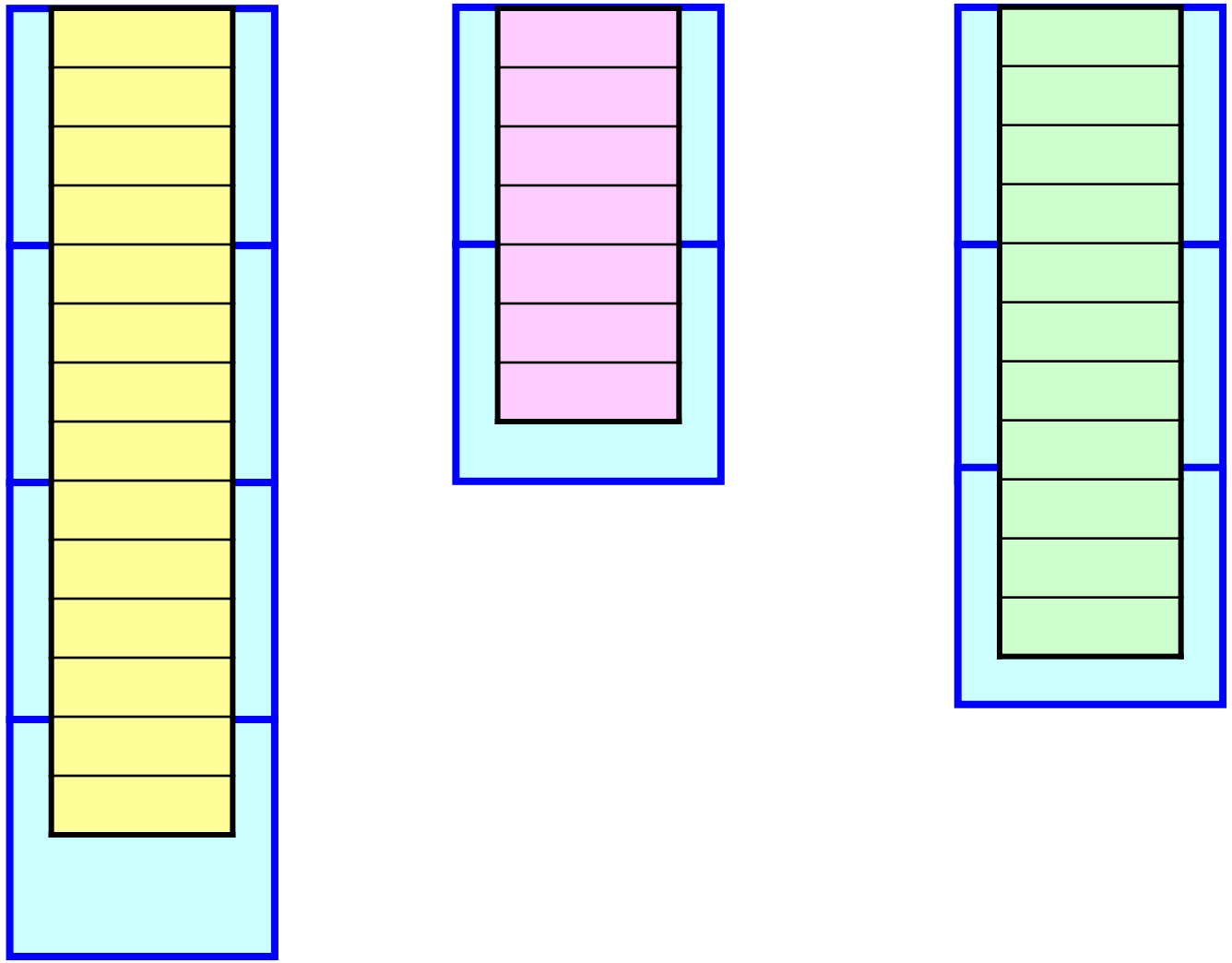
Segmentation Architecture

- **Protection**
  - With each entry in segment table associate:
    - validation bit = 0 illegal segment
    - read/write/execute privileges

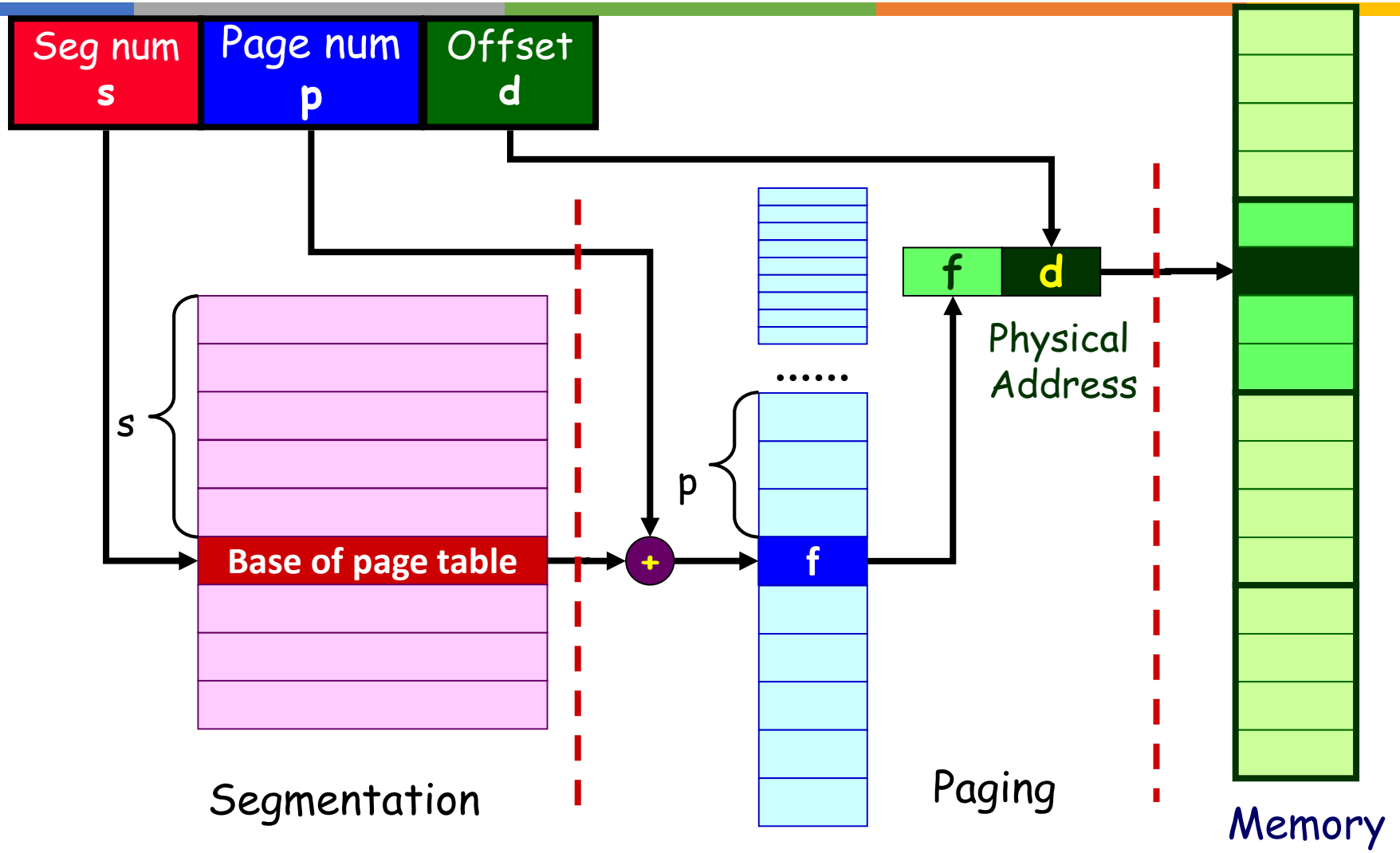
Segmentation with Paging

- Divide each segment into blocks of same size.

Segmentation with Paging



Segmentation with Paging

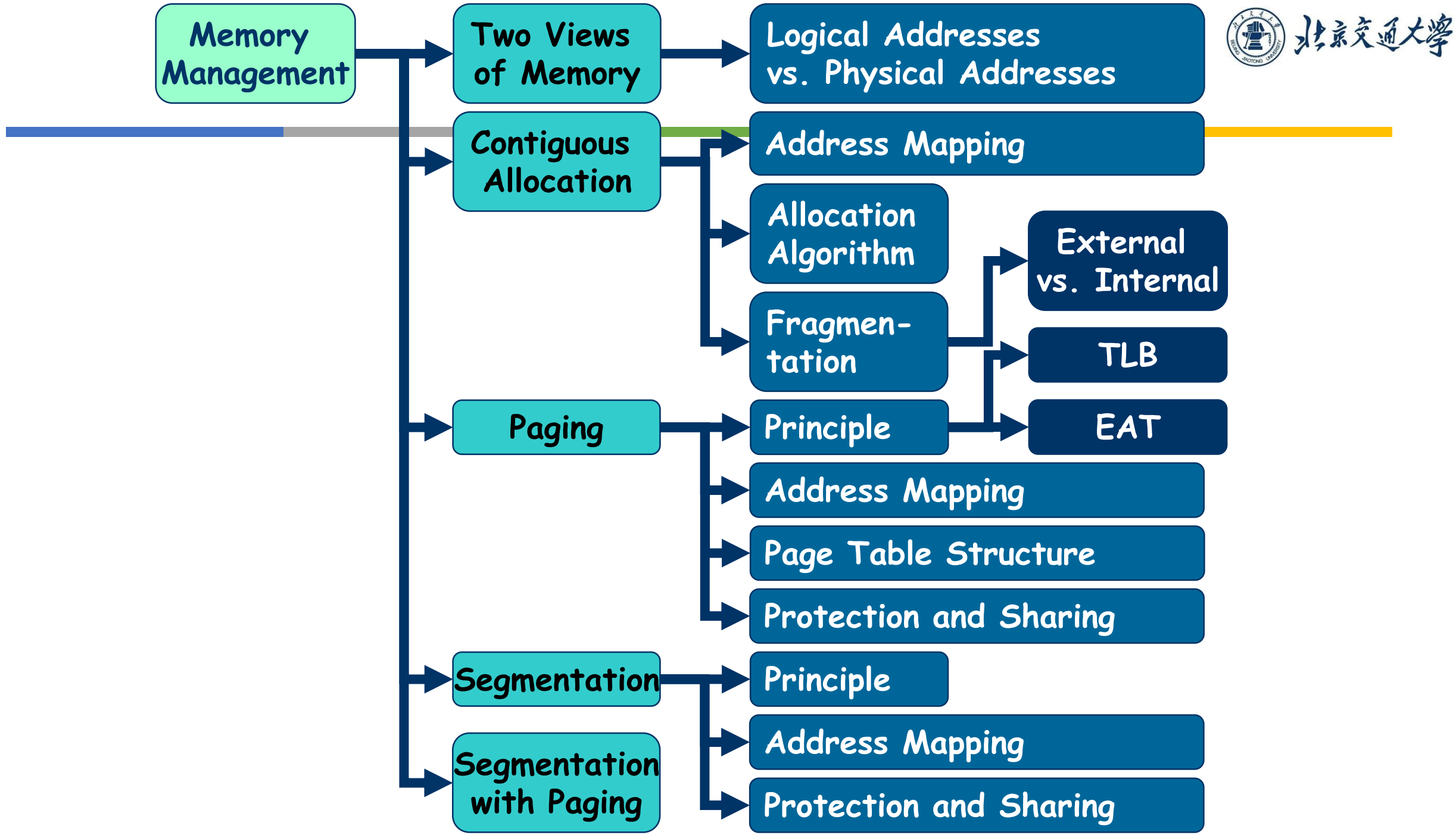


Summary

- Memory is a resource that must be shared
  - Translation: Change Logical Addresses into Physical Addresses
  - Protection: Prevent unauthorized Sharing of resources
- Simple contiguous memory allocation
  - Base+limit registers restrict memory accessible to user
  - Can be used to translate as well
- Full translation of addresses through Memory Management Unit (MMU)

Summary

- Page Tables
  - Memory divided into fixed-sized chunks of memory
  - Logical page number from logical address mapped through page table to physical page number
  - Offset of logical address same as physical address
- Multi-Level Tables
  - Logical address mapped to series of tables
- Hashed page table
- Inverted page table
- Segment Mapping
  - Each segment contains base and limit information


Memory Management
Two Views of Memory
Logical Addresses vs. Physical Addresses
Contiguous Allocation
Address Mapping
Allocation Algorithm
External vs. Internal
Fragmen-tation
TLB
Paging
Principle
EAT
Address Mapping
Page Table Structure
Protection and Sharing
Segmentation
Principle
Address Mapping
Segmentation with Paging
Protection and Sharing

北京交通大学

Thank you!

Q & A