TDDB68/TDDE47
Concurrent Programming and Operating Systems

Lecture:
Memory management I

Mikael Asplund
Real-time Systems Laboratory
Department of Computer and Information Science

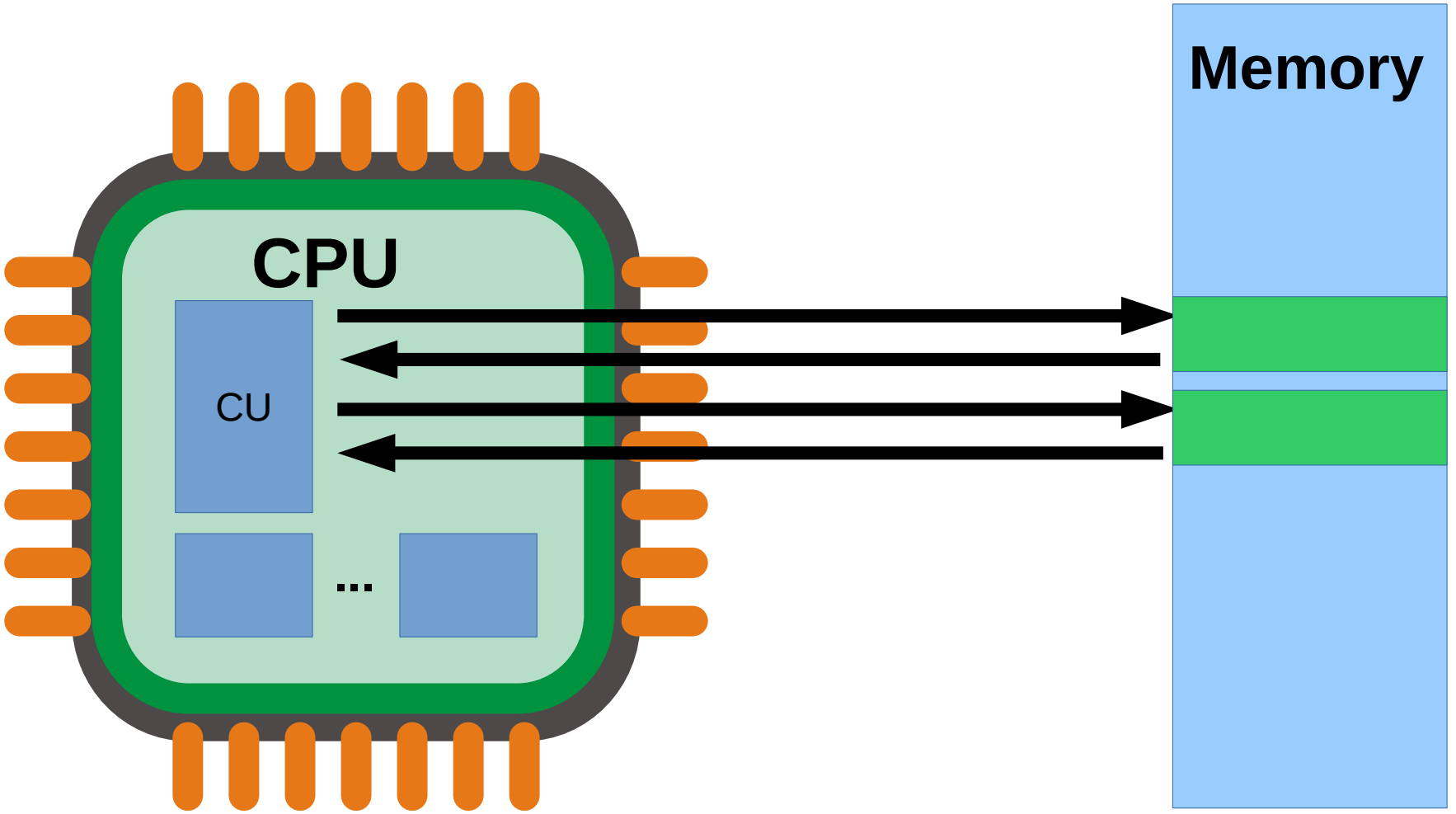
Thanks to Simin Nadjm-Tehrani and Christoph Kessler for much of the material behind these slides.

Reading guidelines

- Sliberschatz et al.
  - 9th edition: Ch. 8, 9.1-9.3
  - 10th edition: Ch. 9, 10.1-10.3

Why do we need memory management?

Simple microprocessor

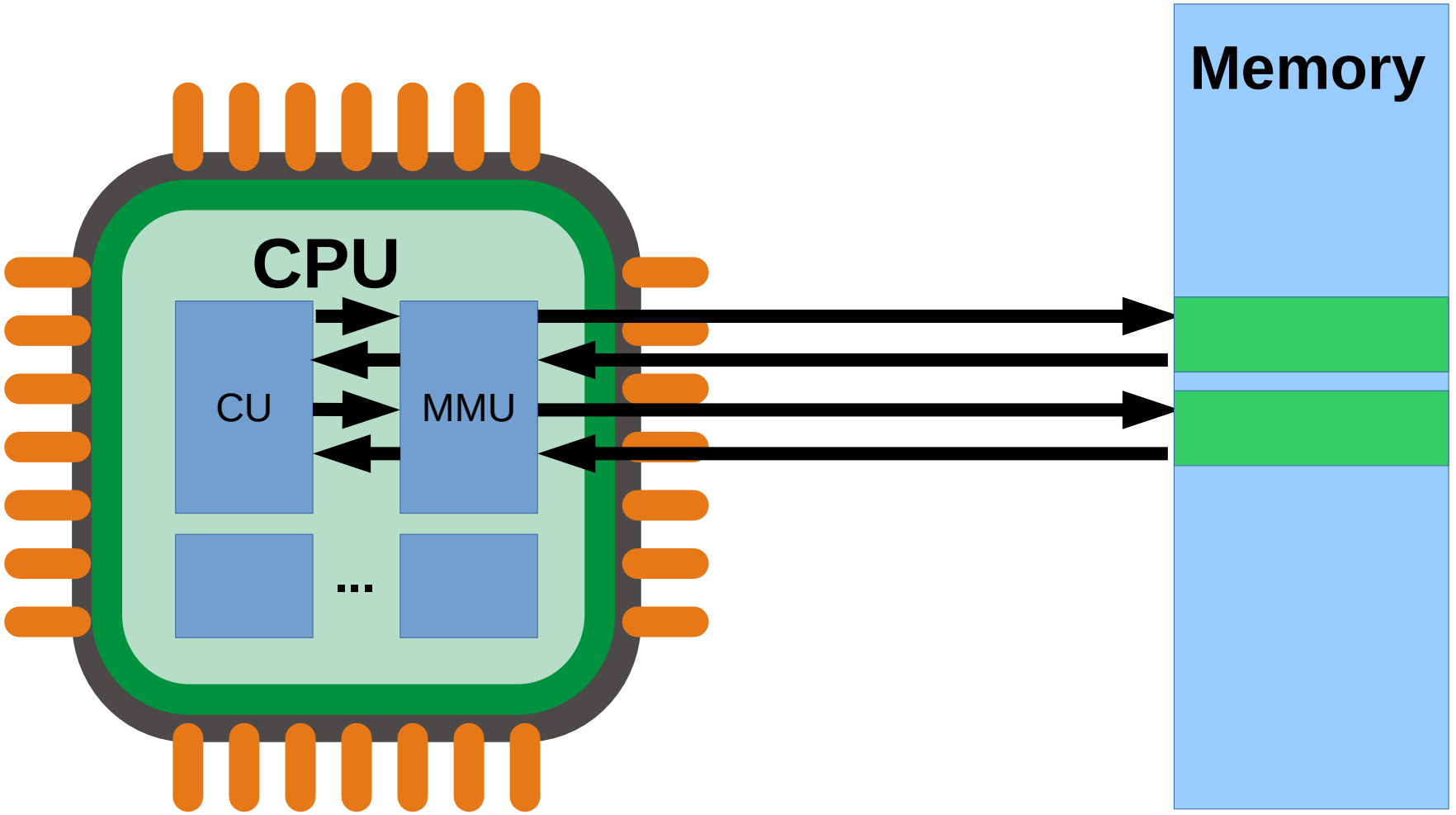


CU = Control Unit

Problems

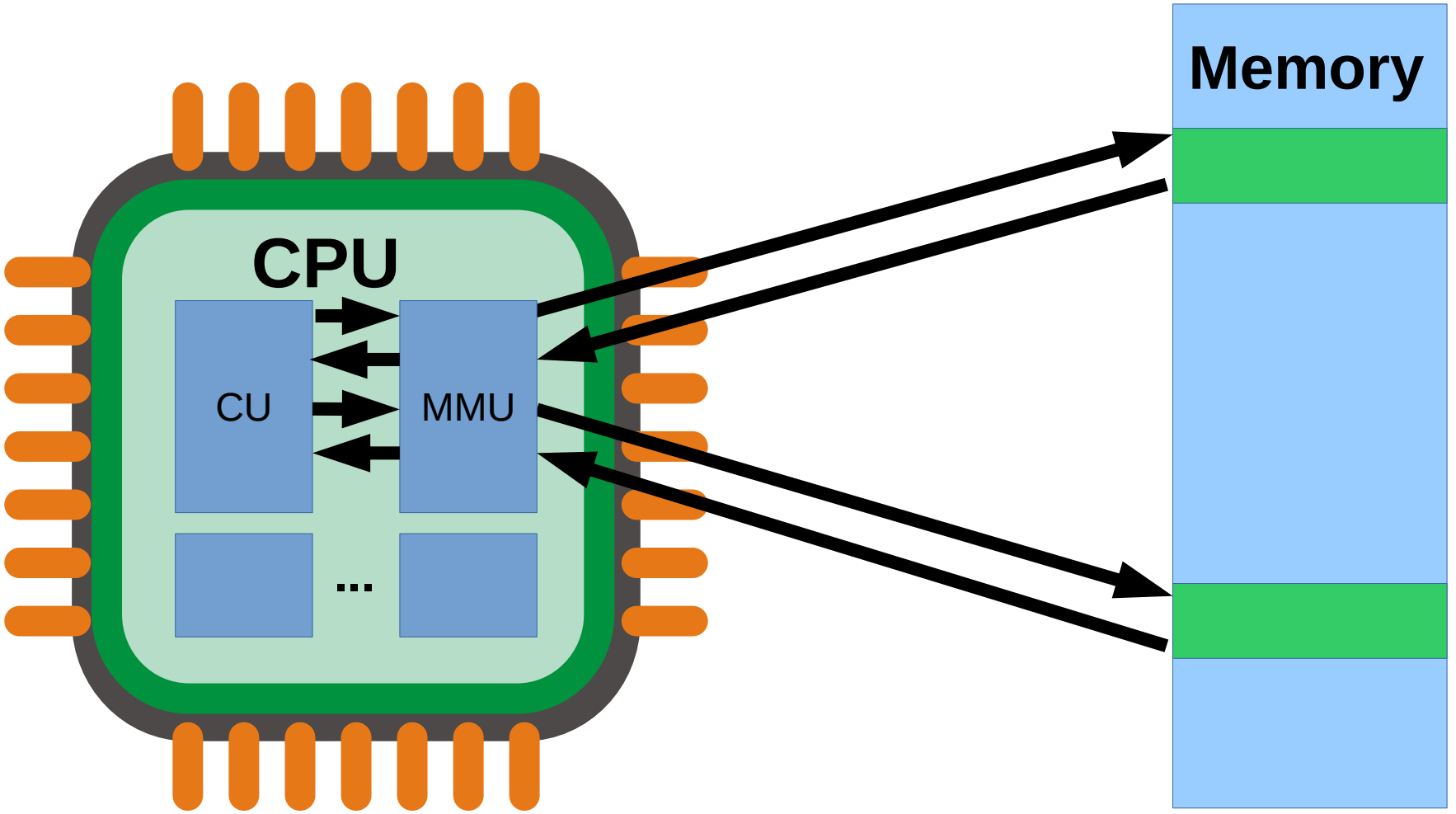
- Process separation
- Dynamic processes
- Memory allocation

High-end CPU



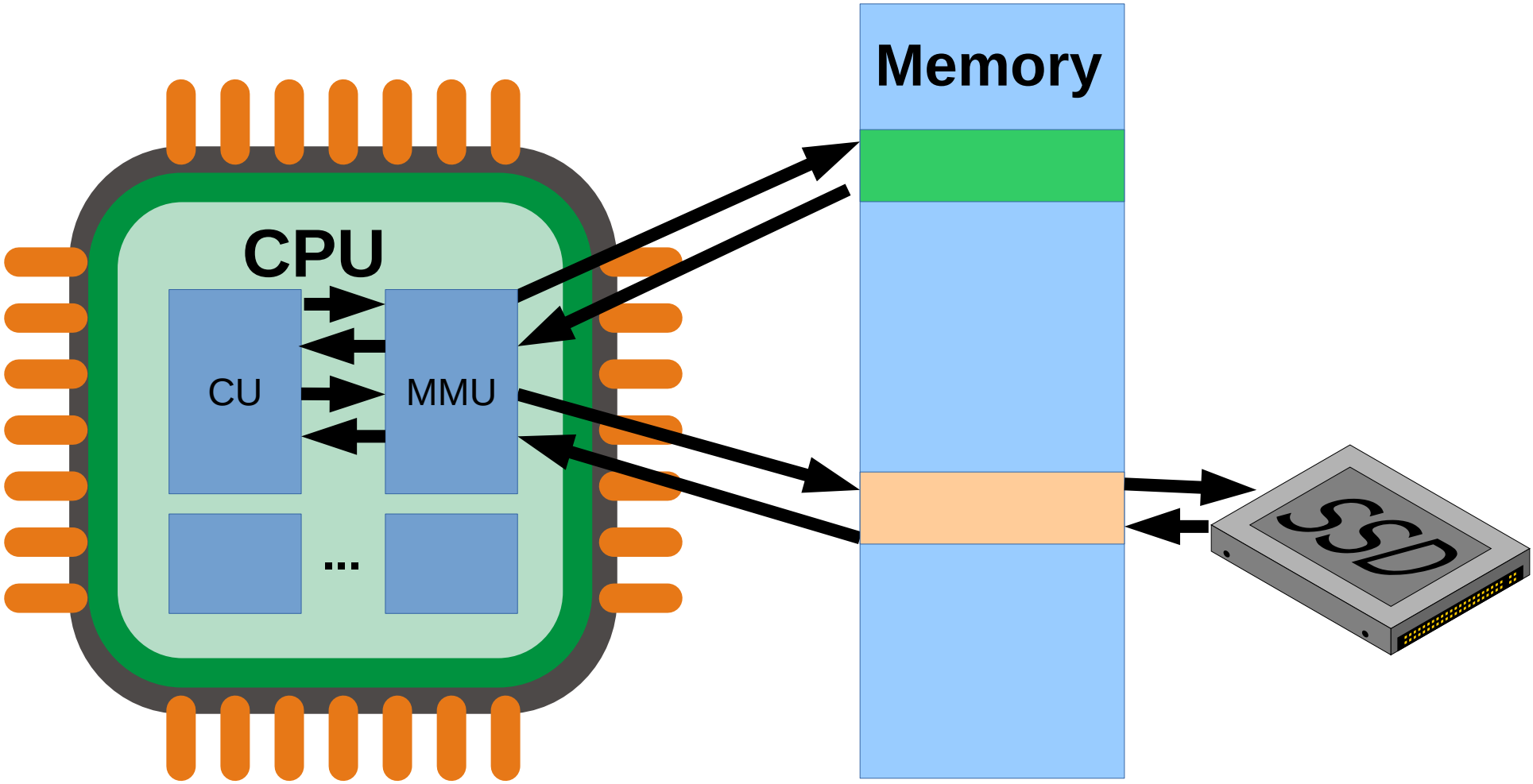
CU = Control Unit, MMU = Memory Management Unit

Logical address != physical address



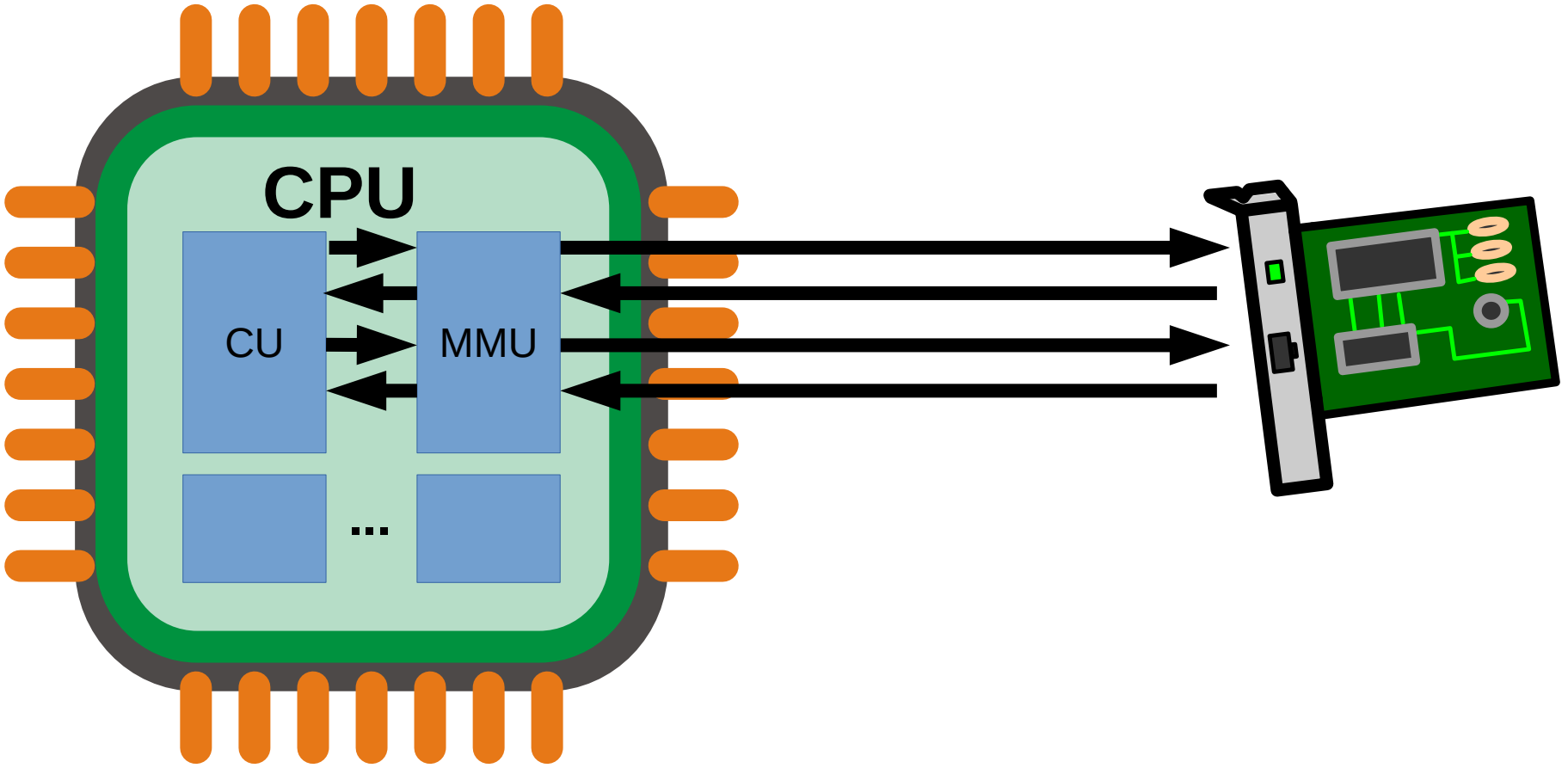
CU = Control Unit, MMU = Memory Management Unit

Can put data elsewhere



CU = Control Unit, MMU = Memory Management Unit

Does not even have to be real memory



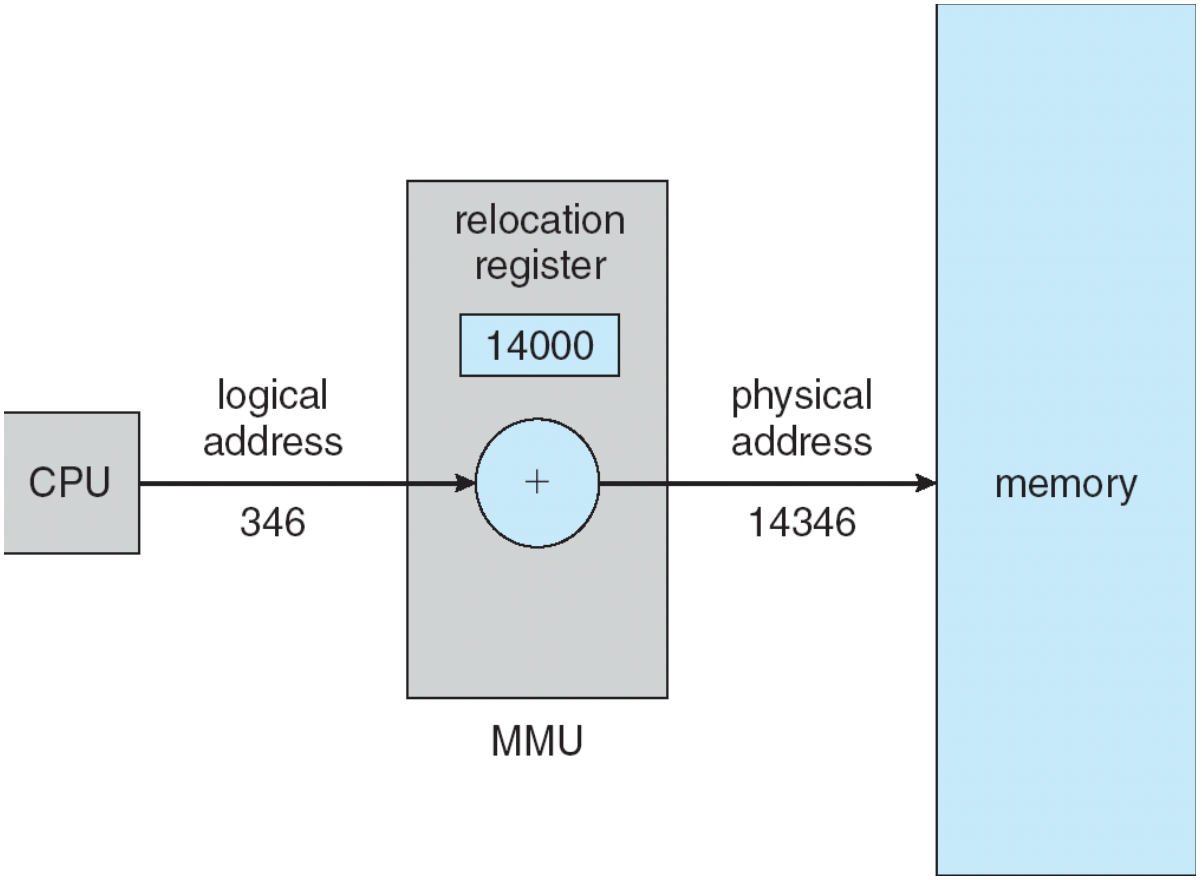
CU = Control Unit, MMU = Memory Management Unit

Due to the separation between logical address and physical address the MMU can provide the process with **virtual memory**

Memory-Management Unit (MMU)

- Hardware device that maps virtual to physical address
- The user program deals with *logical* addresses; it never sees the *real* physical addresses
- The MMU provides translation between logical and physical addresses

Simple MMU idea

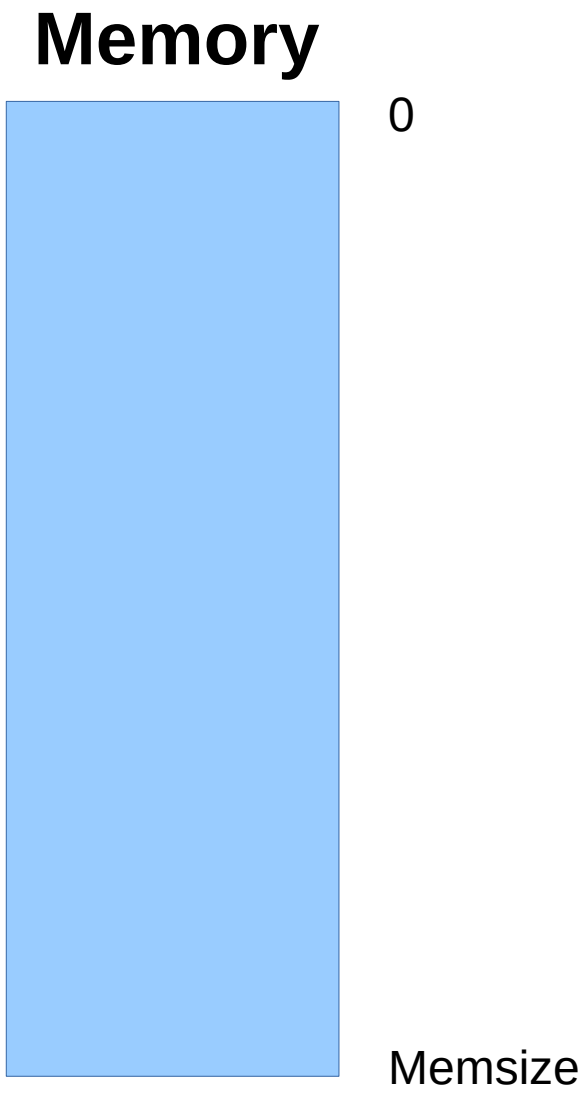


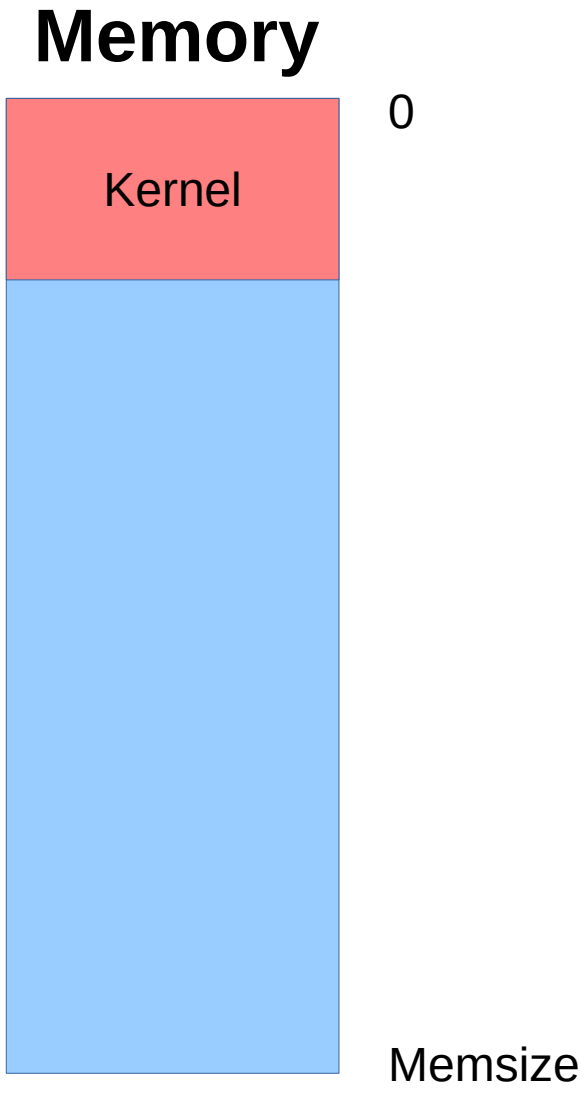
Dynamic relocation using a ***relocation register***

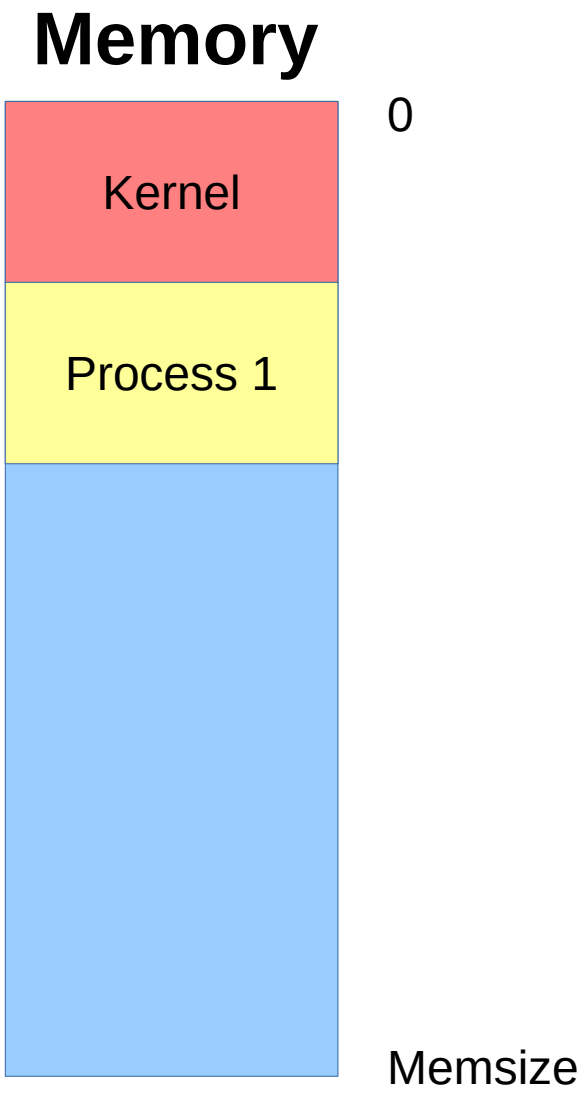
Granularity

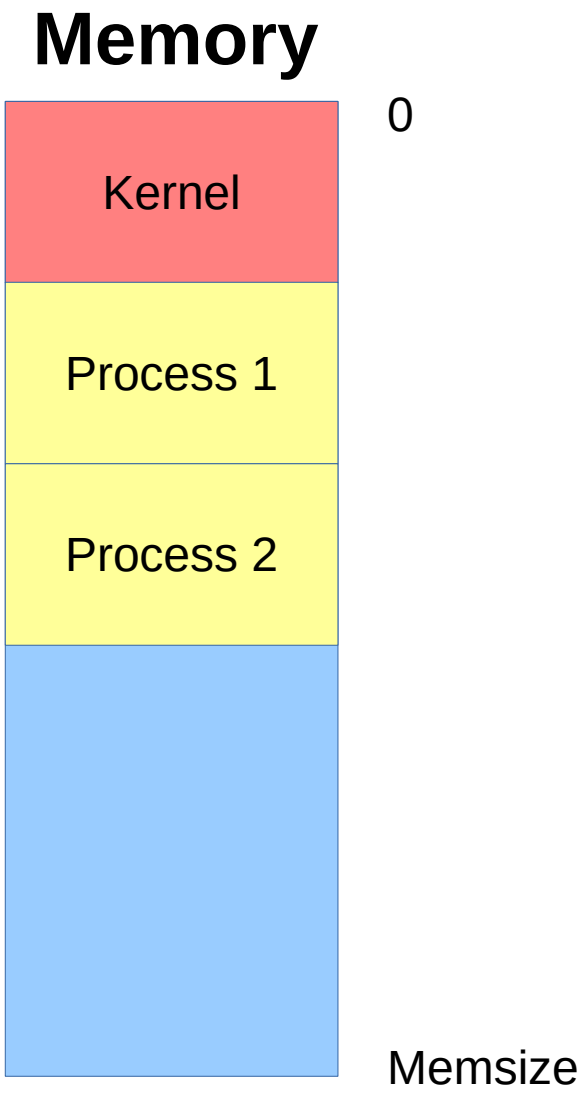
- The ”chunks” of memory that can be given different locations by the MMU can be
  - Process level
  - Segmentation
  - Paging

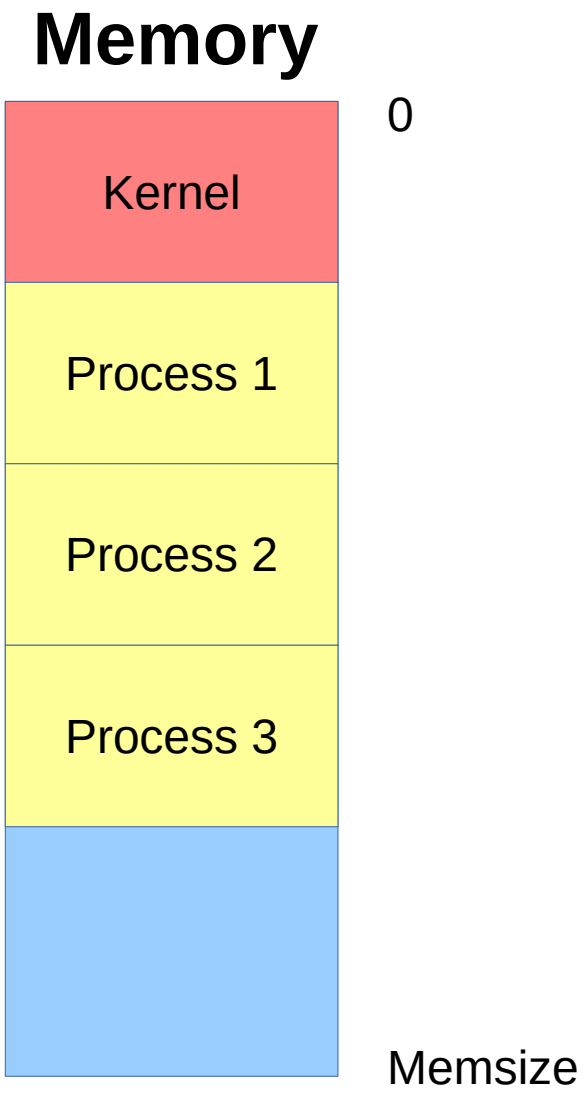
Process-level Memory Management


Memory
0
Memsize


Memory
Kernel
0
Memsize

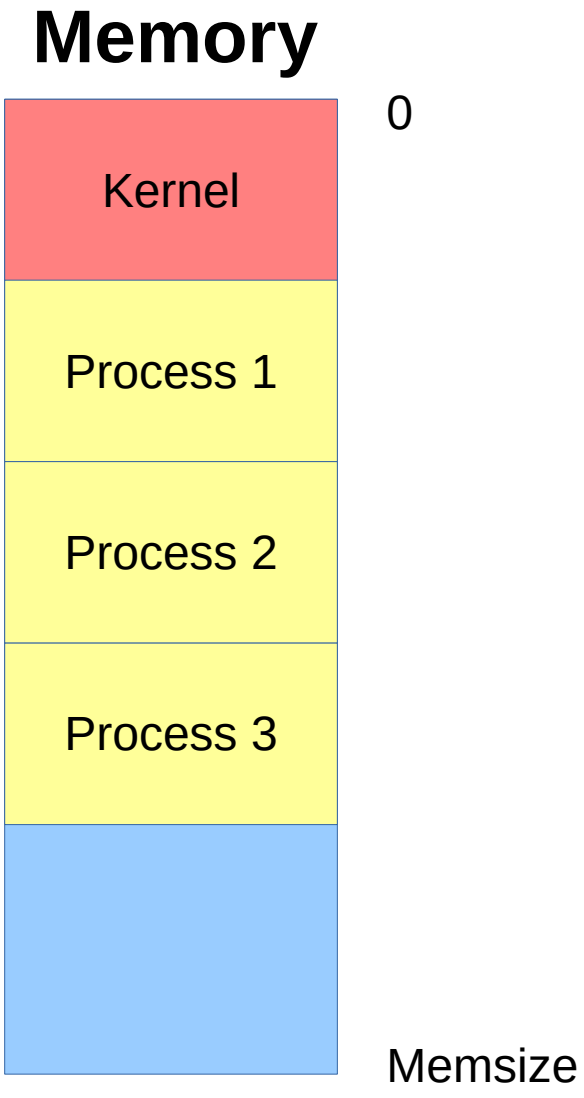

Memory
Kernel
Process 1
0
Memsize


Memory
Kernel
Process 1
Process 2
0
Memsize

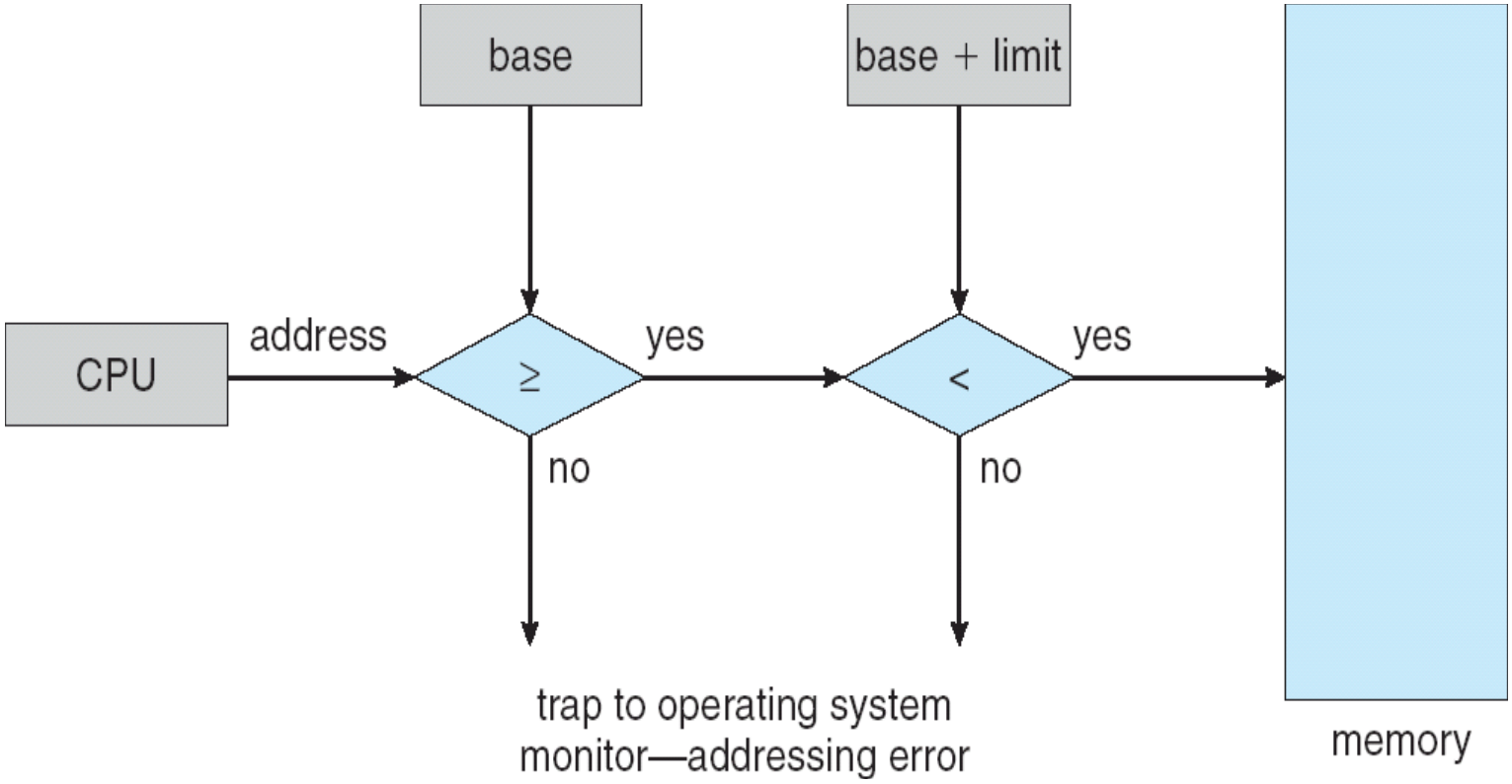

Memory
Kernel
Process 1
Process 2
Process 3
0
Memsize

Multi-partition allocation

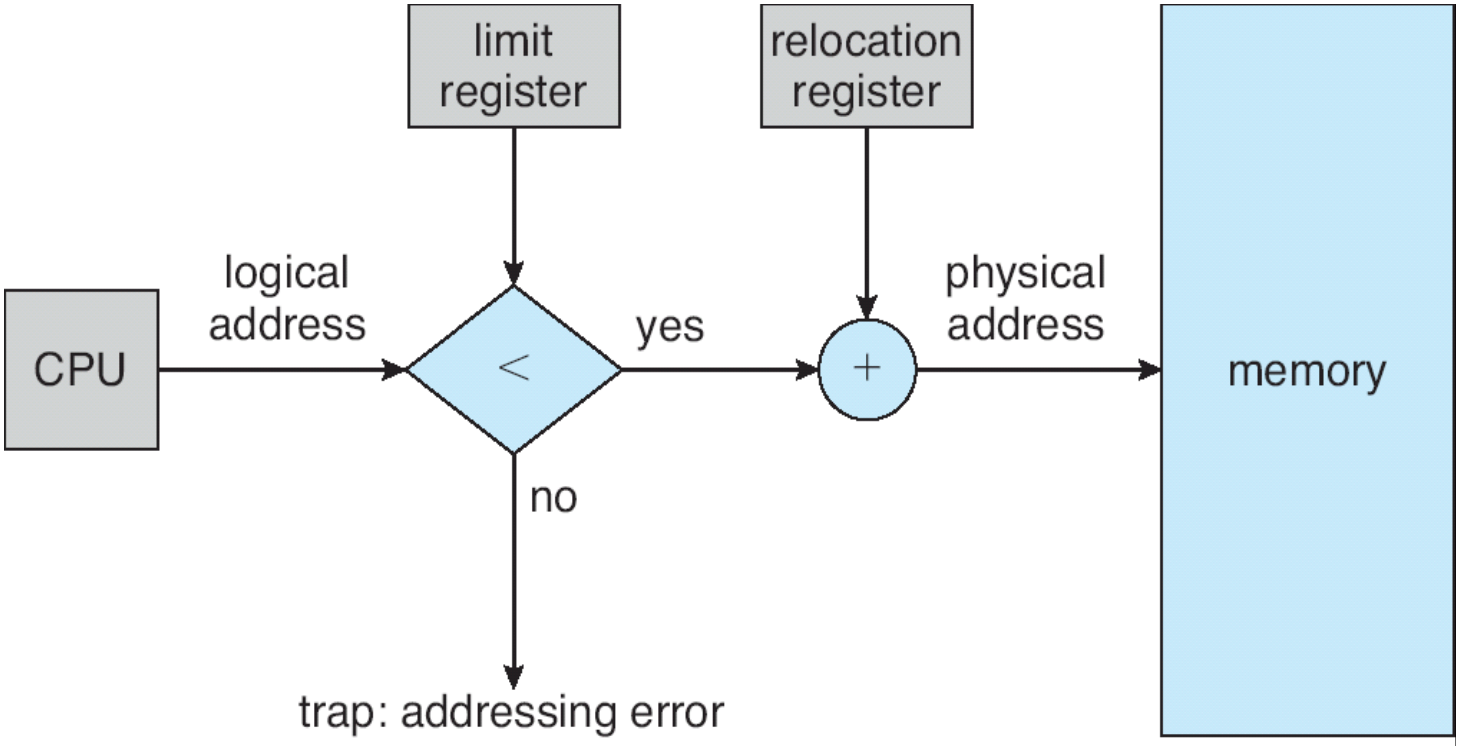
- Each process gets one partition
- Protects processes from each other

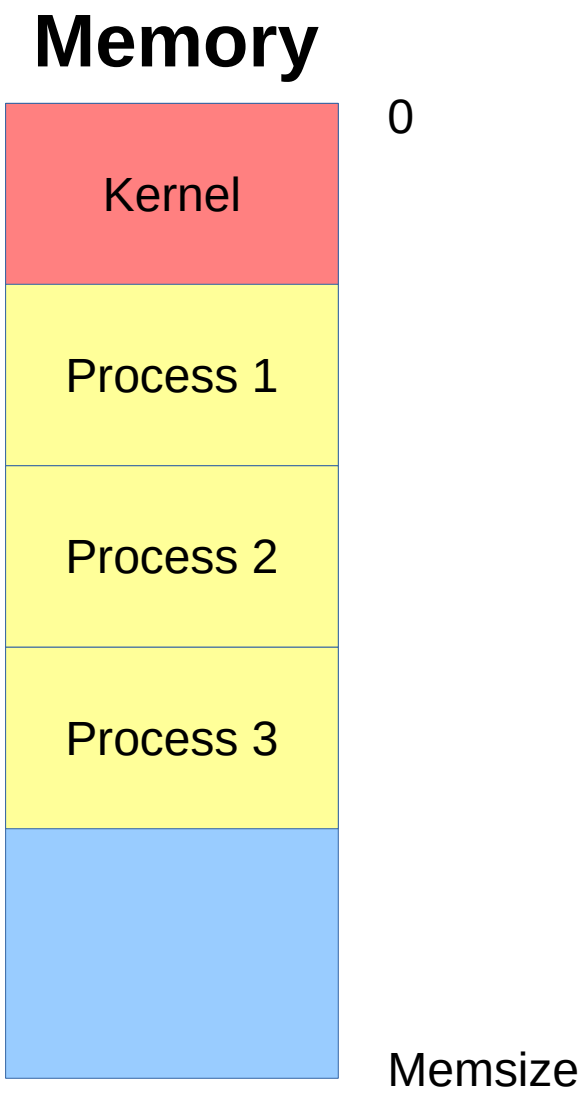


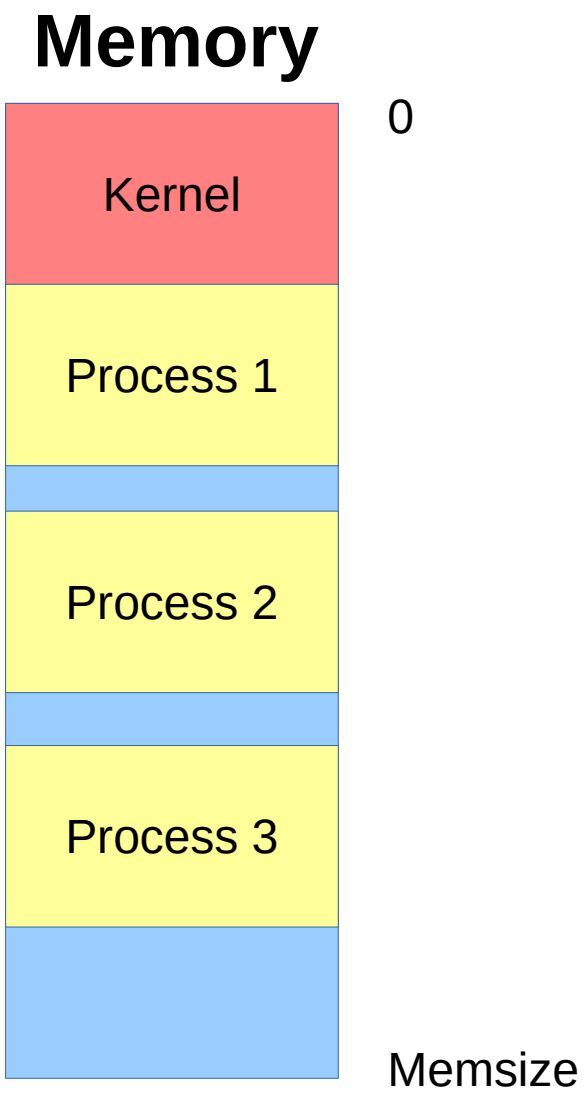
Ensuring protection

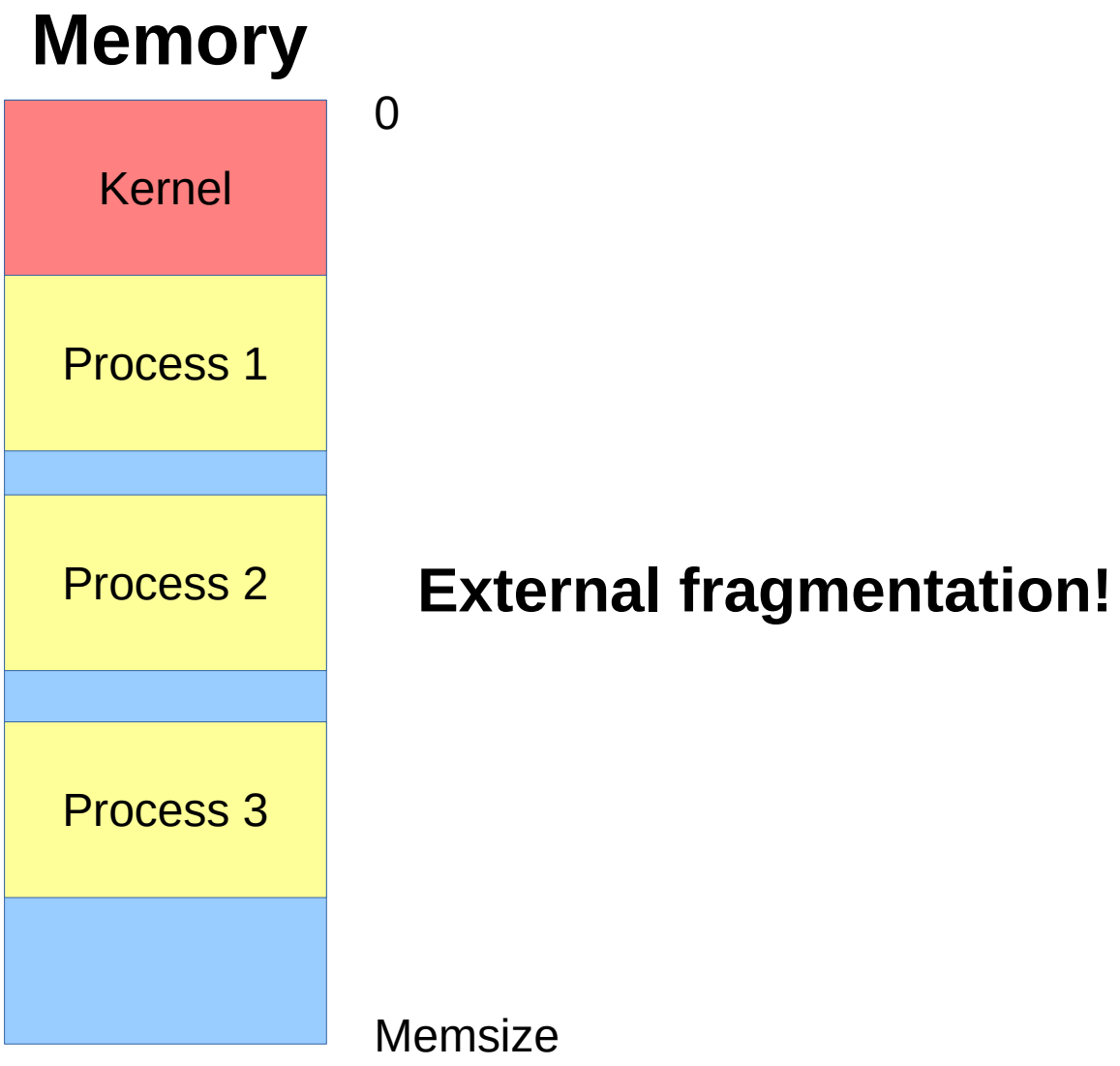


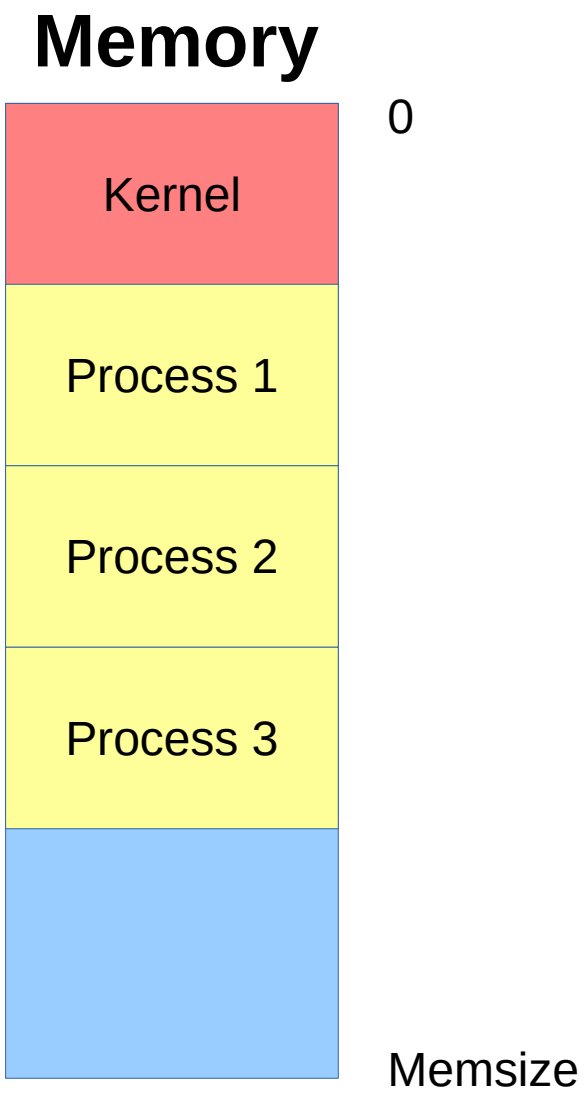
Providing translation

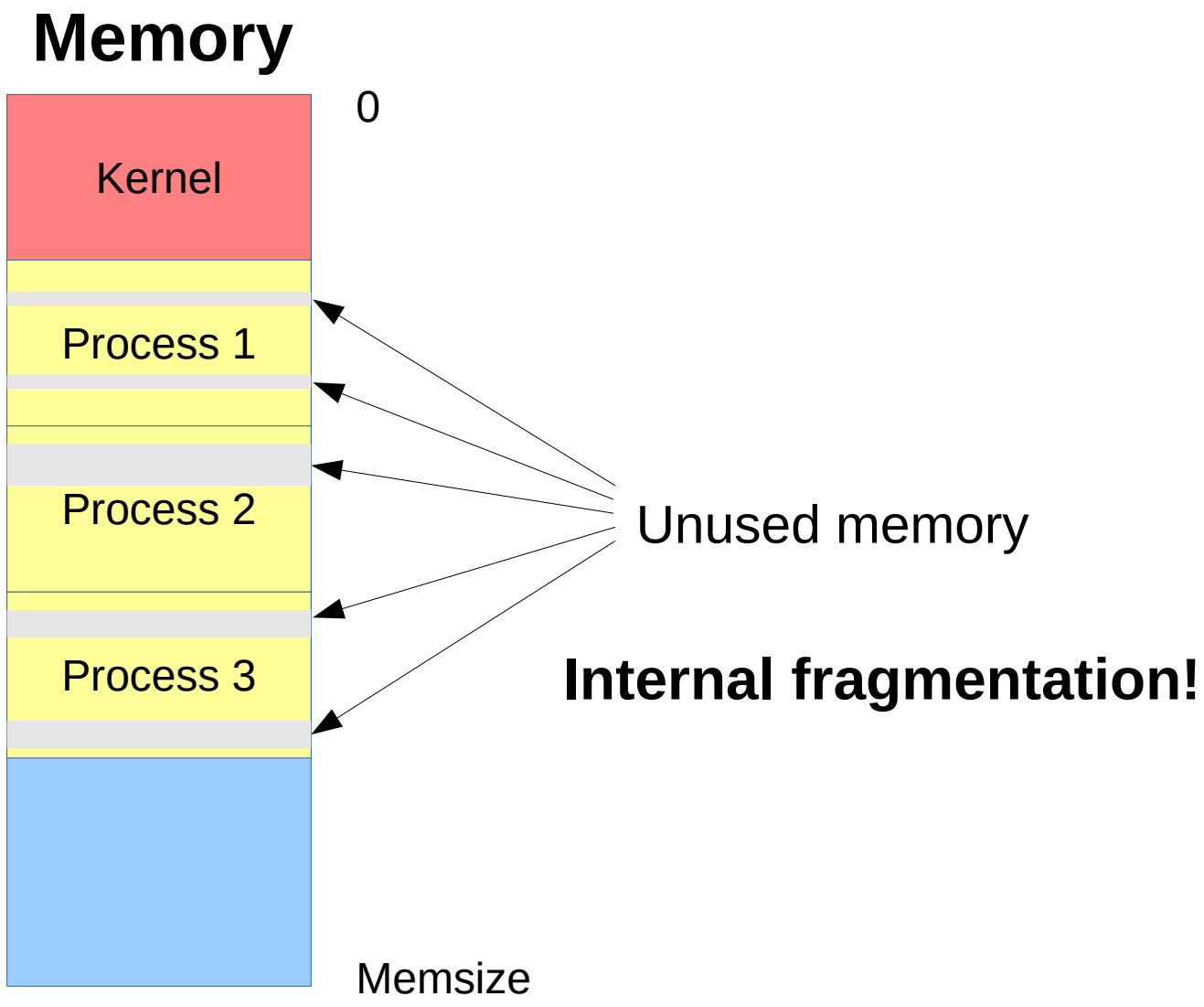



Memory
Kernel
Process 1
Process 2
Process 3
0
Memsize


Memory
Kernel
Process 1
Process 2
Process 3
0
Memsize


Memory
Kernel
Process 1
Process 2
Process 3
0
Memsize
External fragmentation!


Memory
Kernel
Process 1
Process 2
Process 3
0
Memsize


Memory
0
Kernel
Process 1
Process 2
Process 3
Unused memory
Internal fragmentation!
Memsize

When allocating memory to a process, how big should the partition be?

Allocation schemes

Fixed partition size

- One size fits all
- Simple
- Internal fragmentation

Variable partition size

- Size of program decides partition size
- More scalable in number of processes
- External fragmentation

Dynamic storage-allocation problem:

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

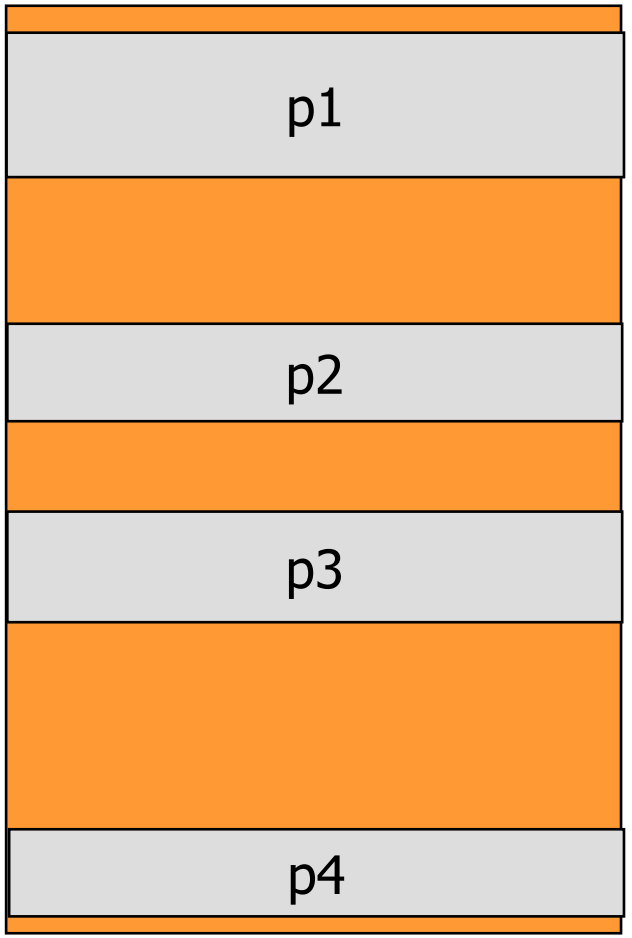
Allocation schemes

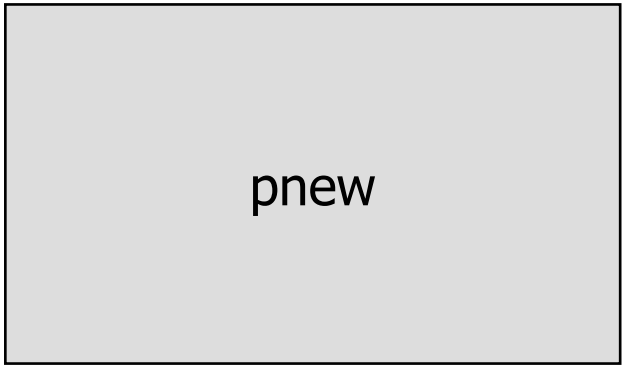
- **First-fit**: Allocate the *first* hole that is big enough
- **Best-fit**: Allocate the *smallest* hole that is big enough;
  - must search entire list, unless ordered by size.
  - Produces the smallest leftover hole.
- **Worst-fit**: Allocate the *largest* hole;
  - must also search entire list.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

Compaction

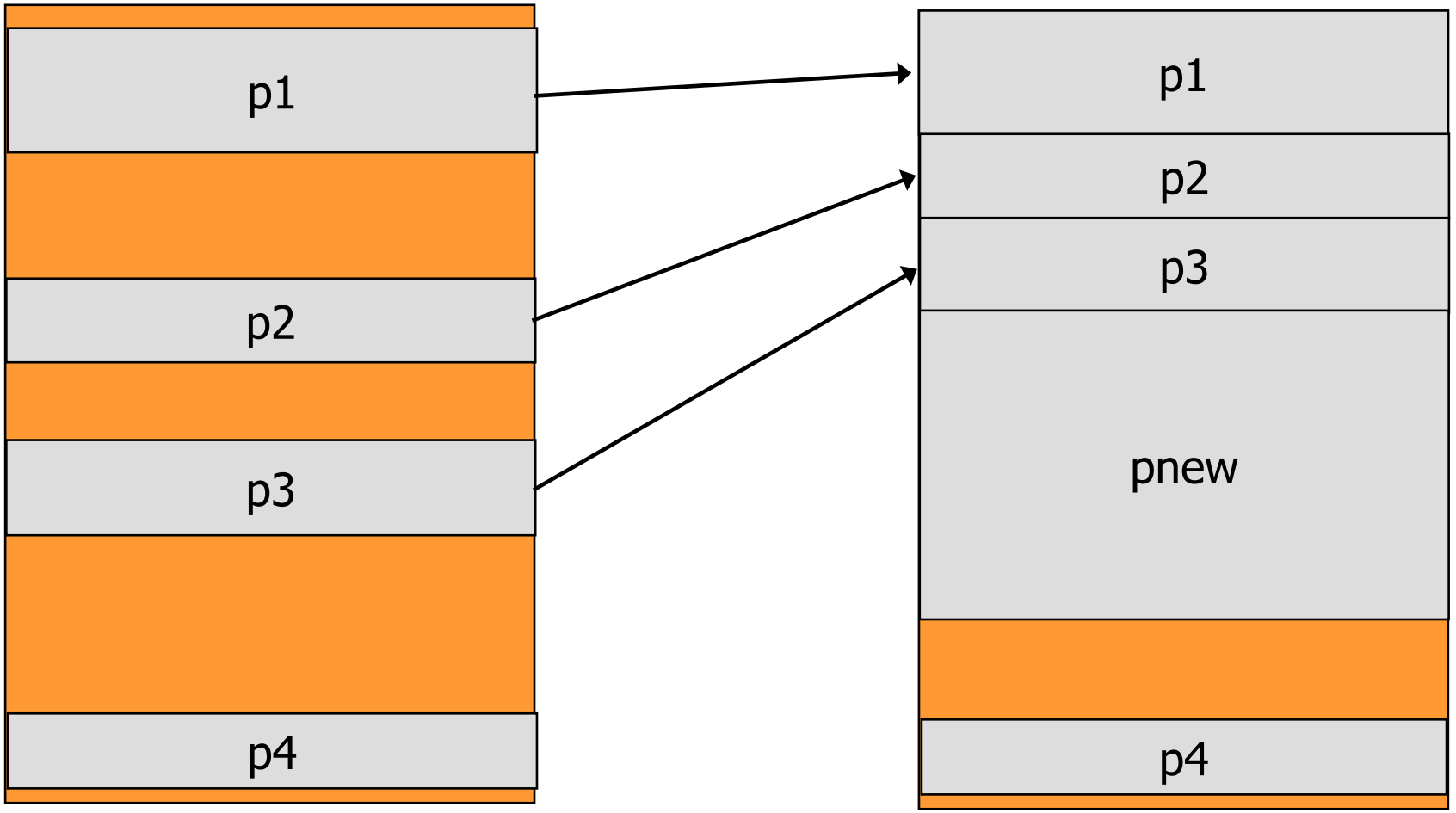
- Reduce external fragmentation
- Compaction is possible *only* if relocation is dynamic, and is done at execution time
- I/O problem

Example of Compacting



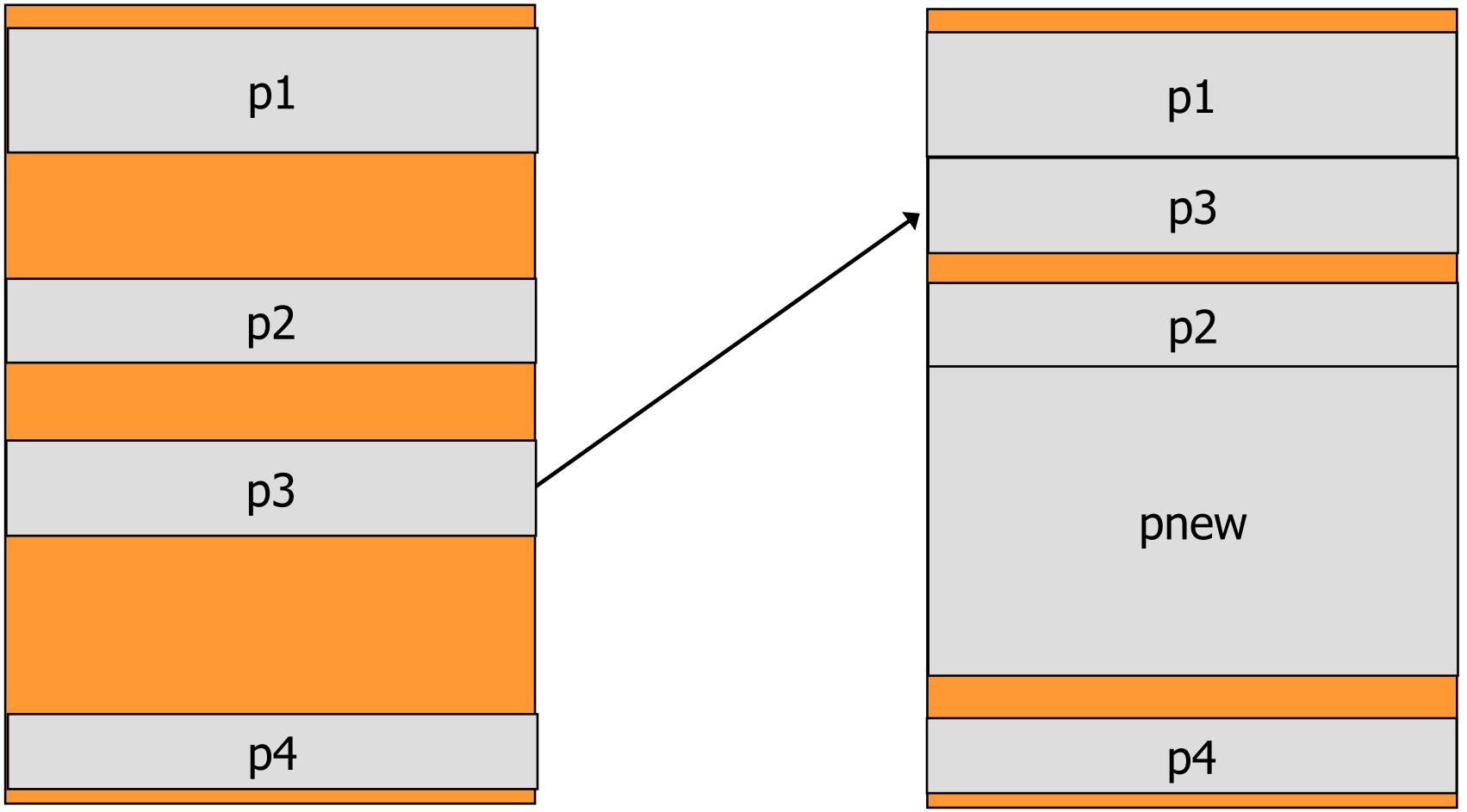


Example of Compacting: Solution 1



Move all occupied areas to one side until there is a hole large enough for pnew

Example of Compacting: Solution 2



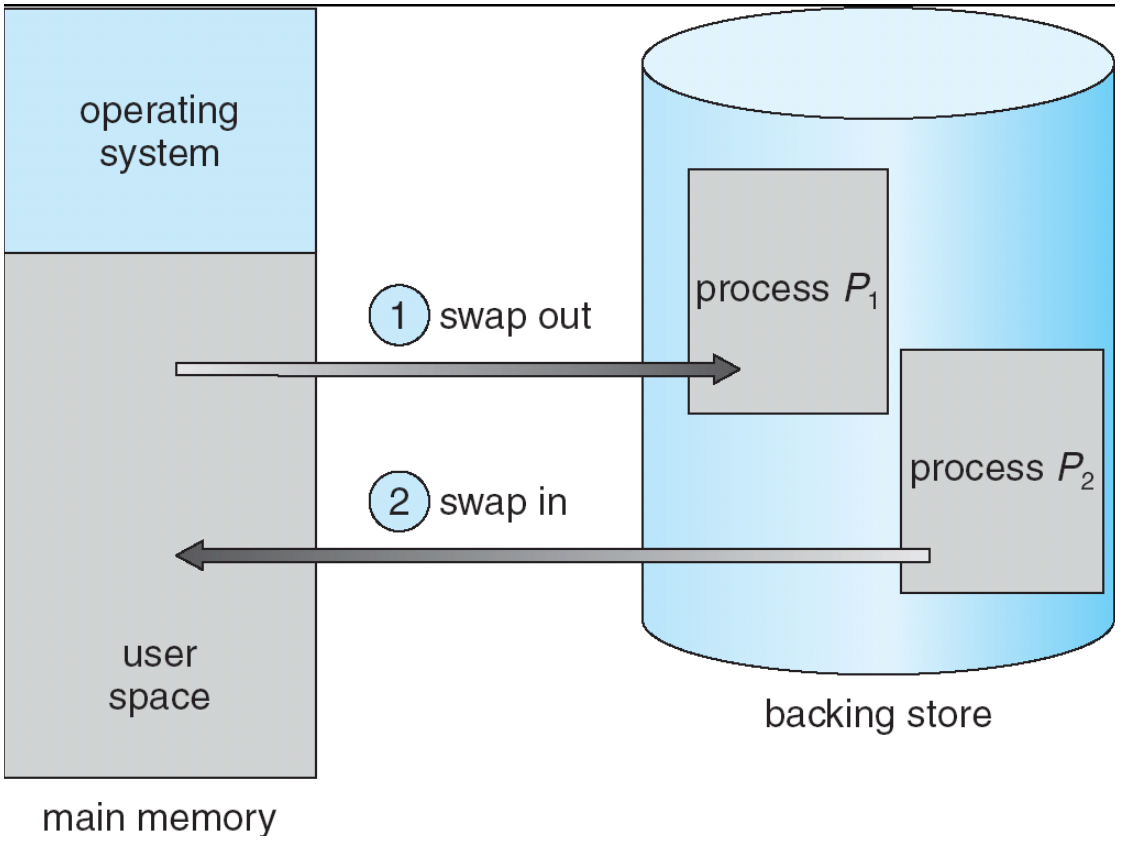
Search and select one (or a few) processes to move to free a hole large enough…

Still not enough space for process?

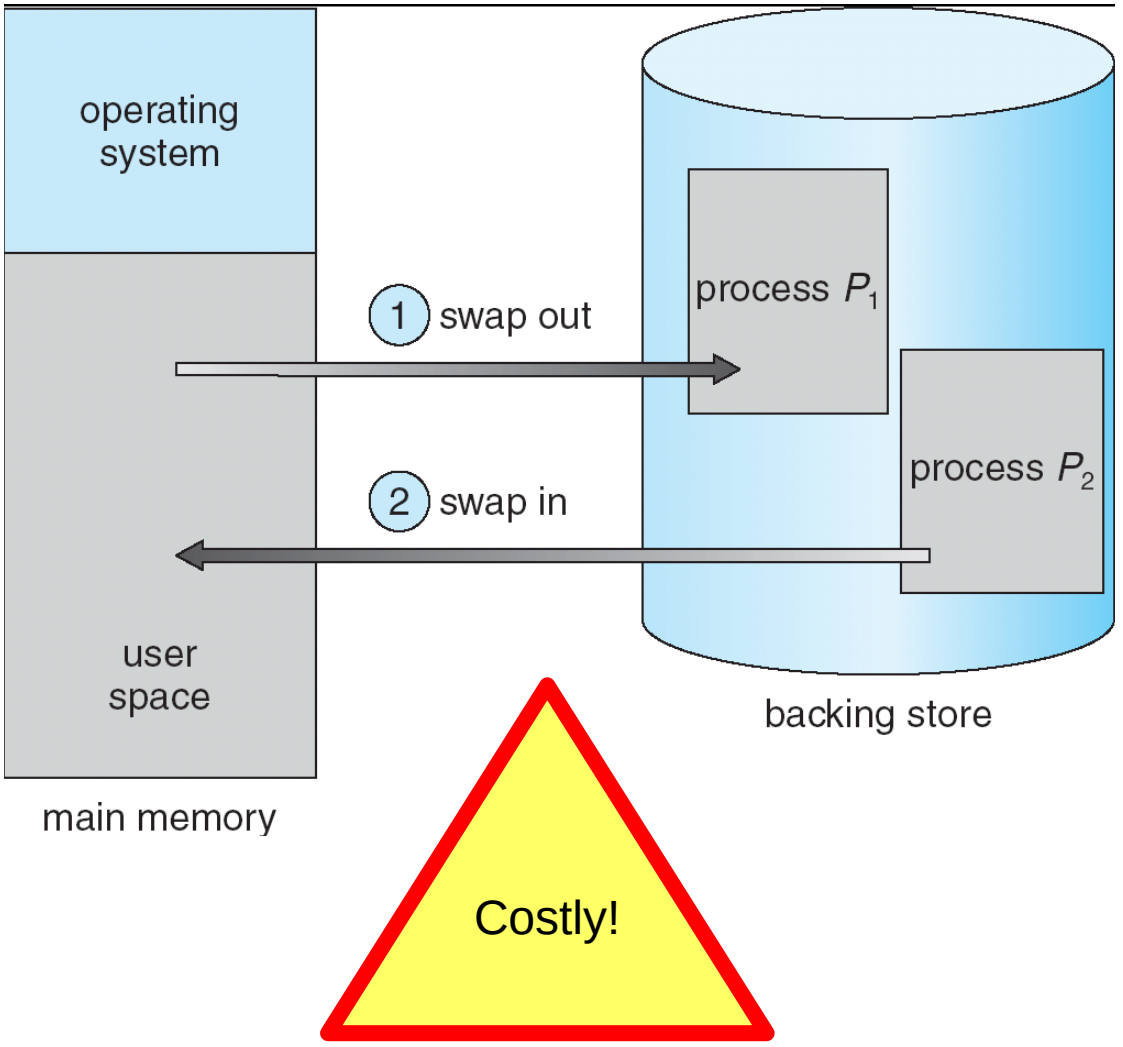
Still not enough space for process?

Try swapping!

Swapping


operating system
user space
main memory
1 swap out
2 swap in
process P_1
process P_2
backing store

Swapping


operating system
user space
main memory
1 swap out
2 swap in
process P_1
process P_2
backing store
Costly!

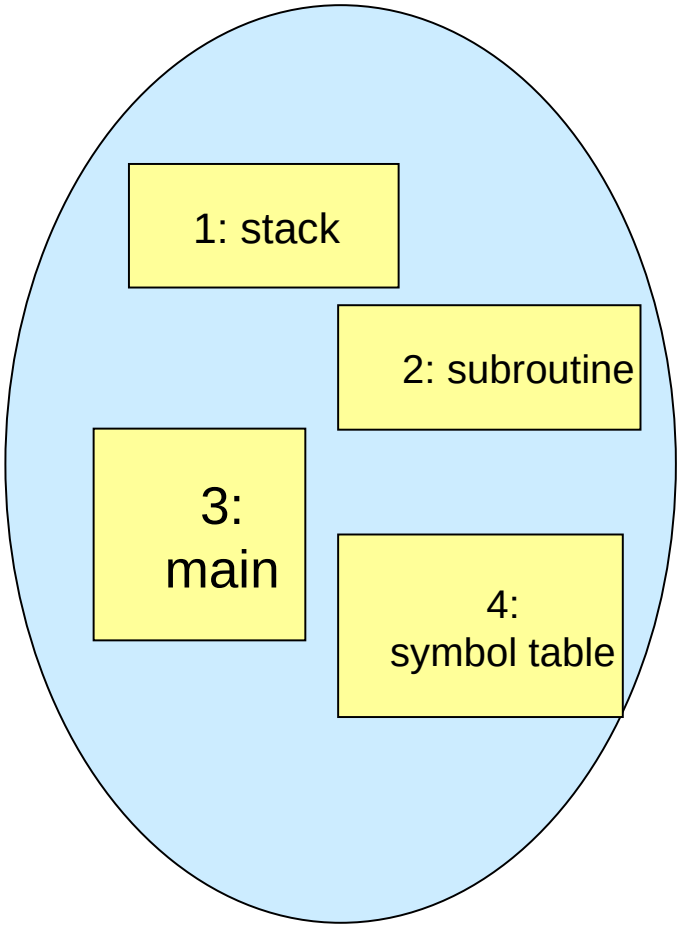
Segmentation

Segmentation

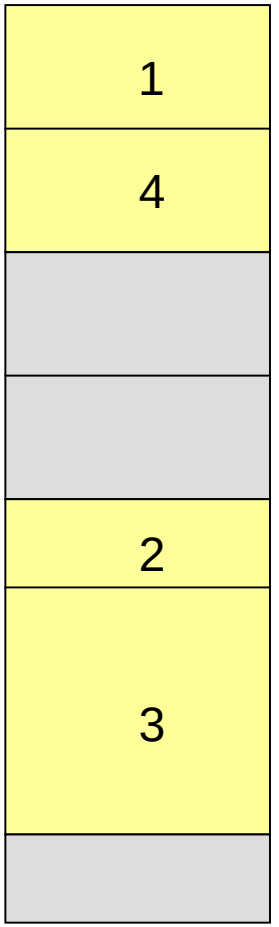
- Memory-management scheme
  that supports a *user view of memory*:

- 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
- Idea: allocate memory according to such segments

Logical View of Segmentation



user space



physical memory space

Segmentation Architecture

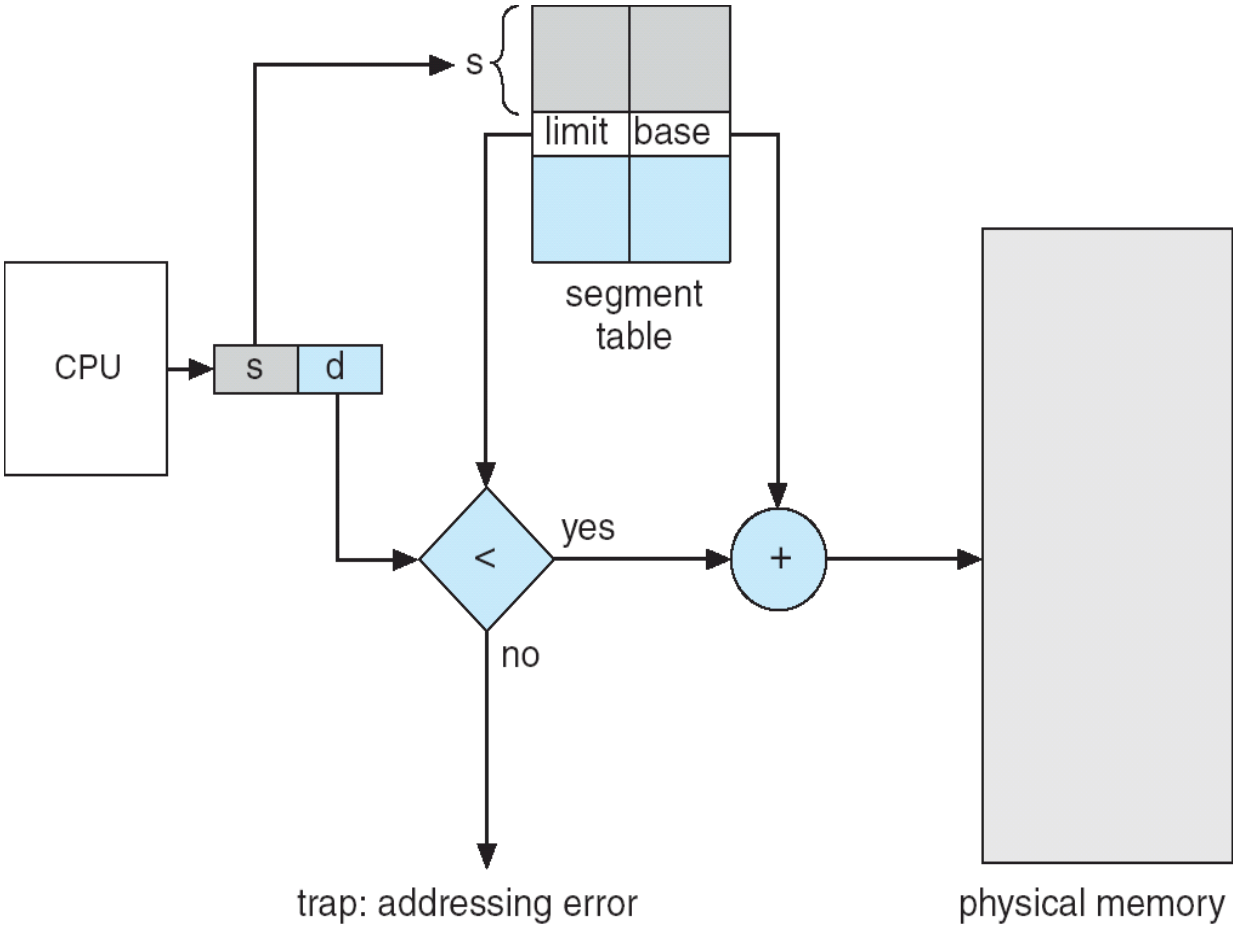
- Logical address is a pair <*segment-number, offset*>

- **Segment table** – maps two-dimensional physical addresses; each table entry has:
  - *base* – physical starting address where the segment resides in memory
  - *limit* – specifies the length of the segment

- 2 registers (part of PCB):
  - *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: Address Translation

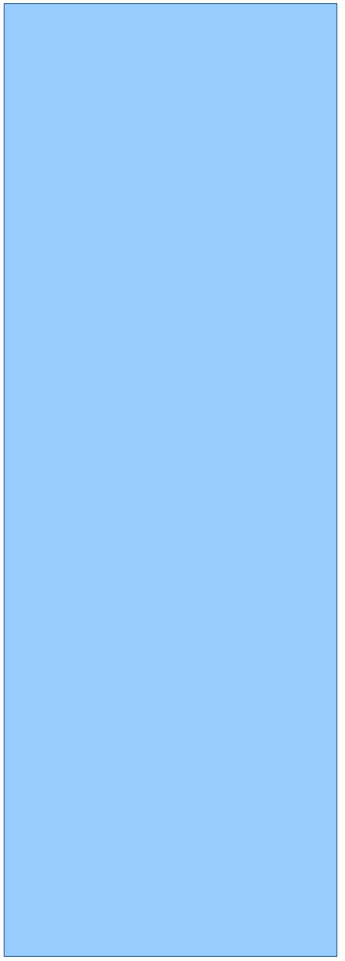


Pros and cons of segmentation

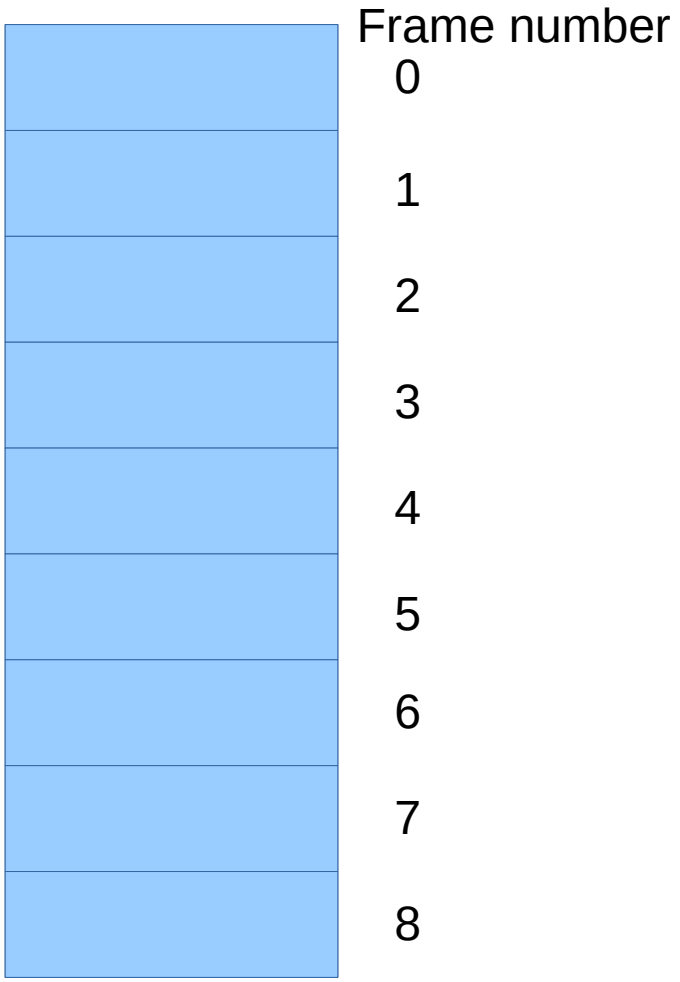
- More fine grained than process-level memory management
- Minimal internal fragmentation
- External fragmentation
- Allocation potentially difficult

Paging

Physical memory



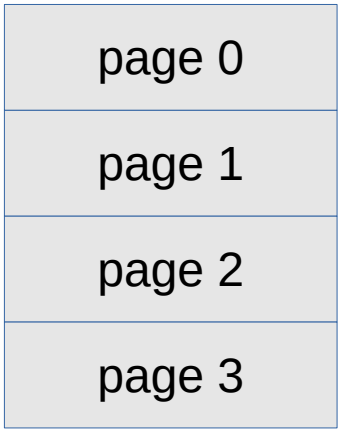
Physical memory



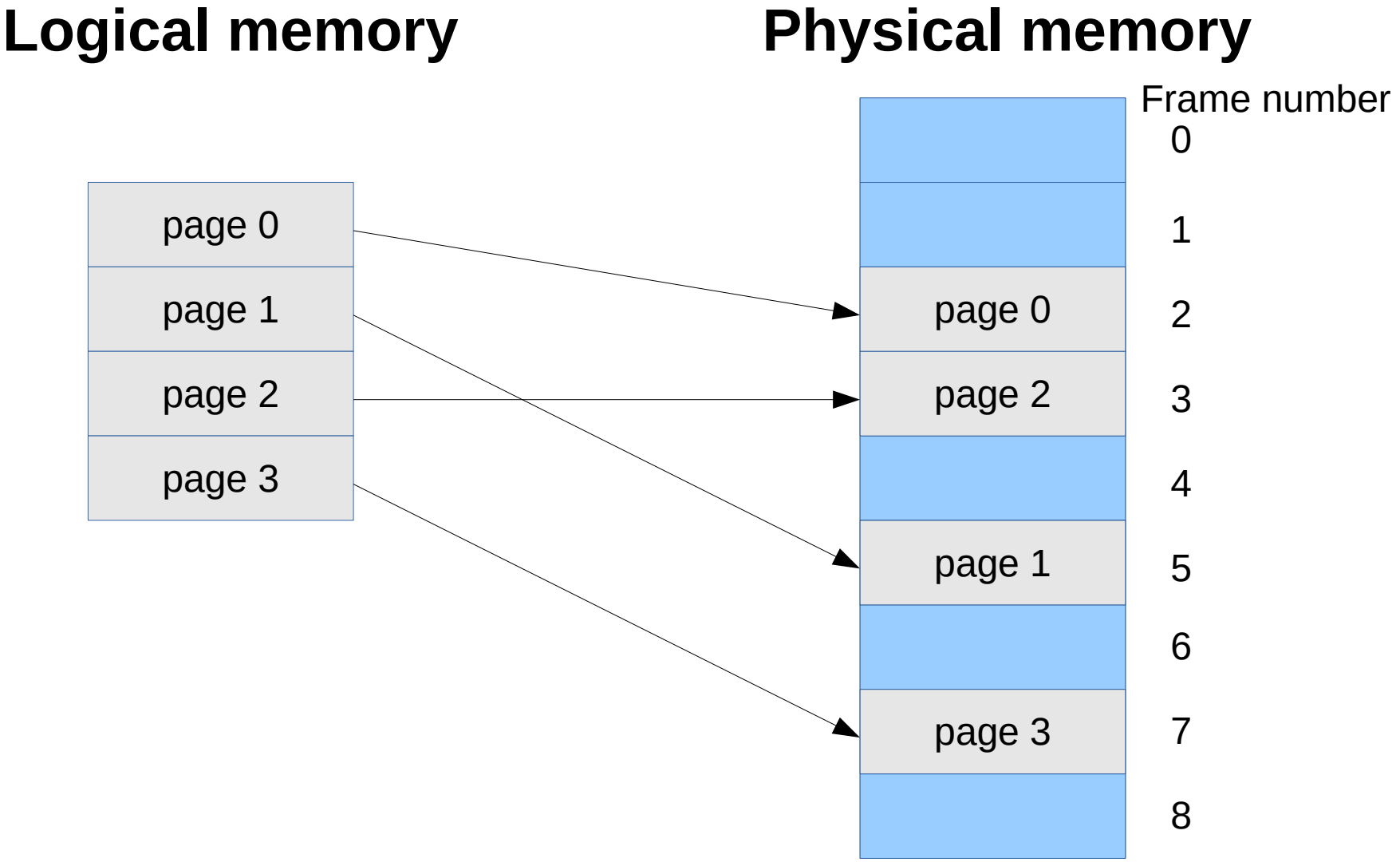
Logical memory

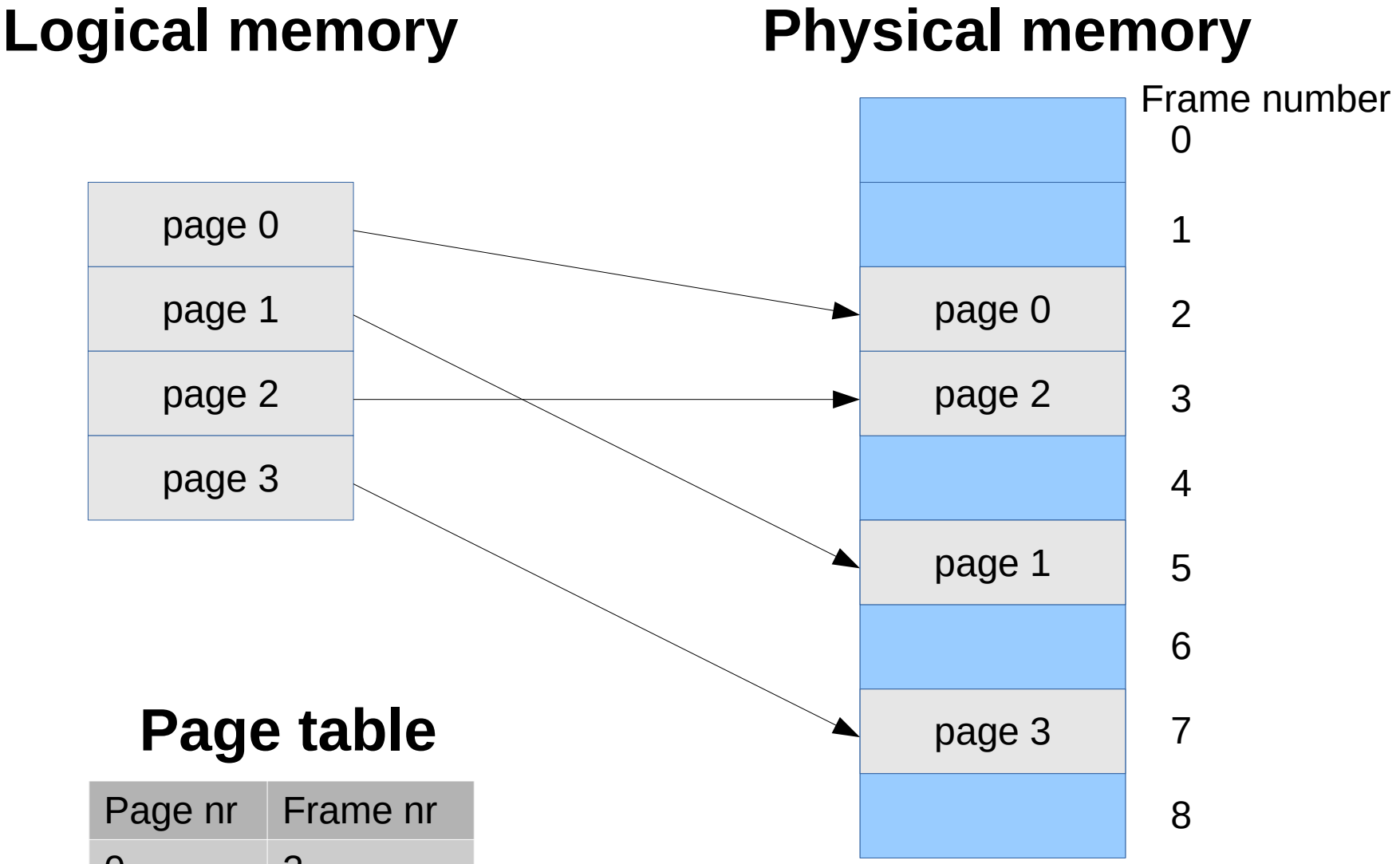
Physical memory

Logical memory



Physical memory


Logical memory
Physical memory
Frame number
page 0
page 1
page 2
page 3
page 0
page 2
page 1
page 3
0
1
2
3
4
5
6
7
8



Page table

Page nr	Frame nr
0	2
1	5
2	3
3	7

Paging

☺ Physical address space of a *process* can be noncontiguous

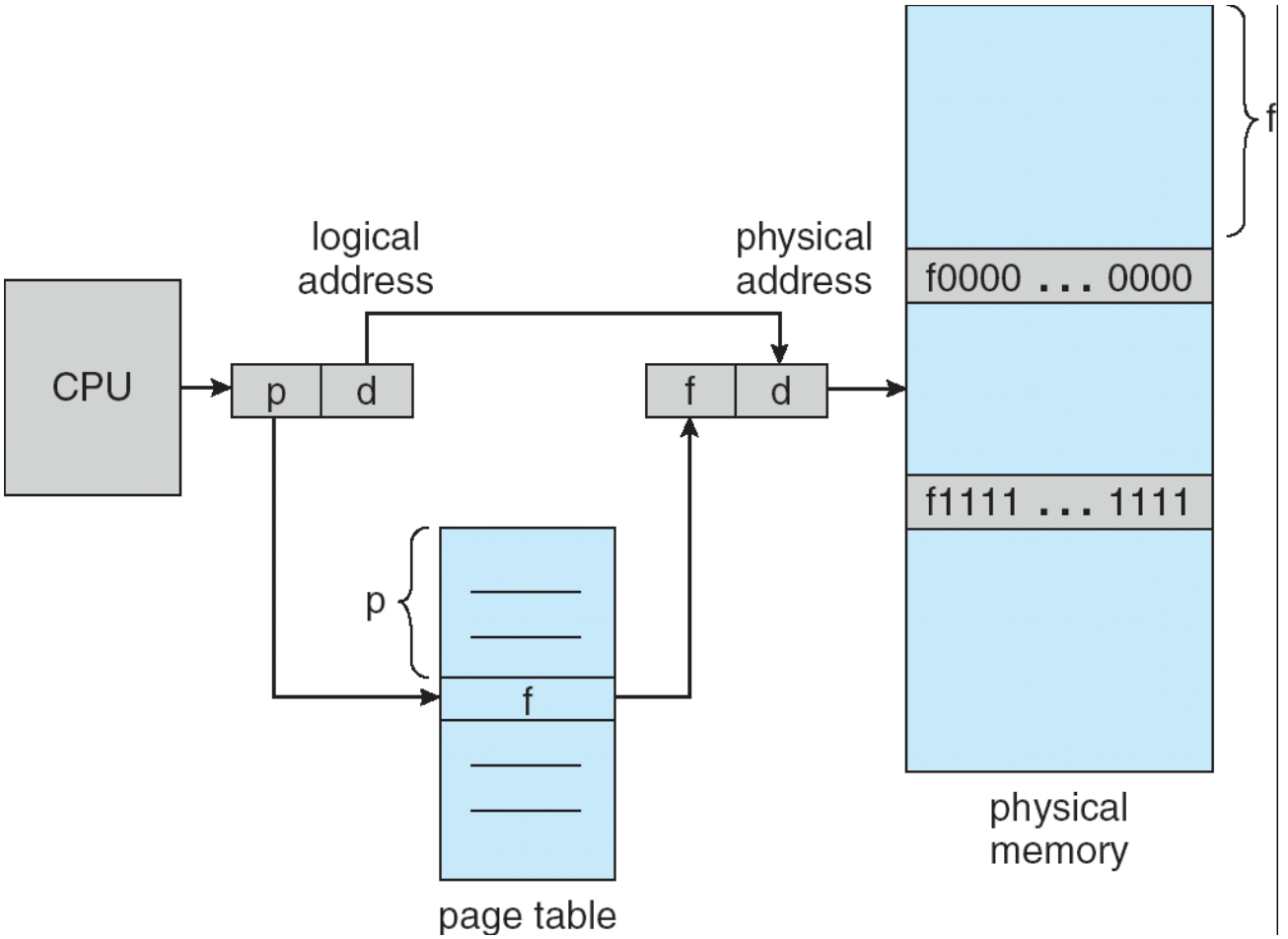
☺ Process is allocated physical memory whenever the latter is available – no external fragmentation

☹ Internal fragmentation

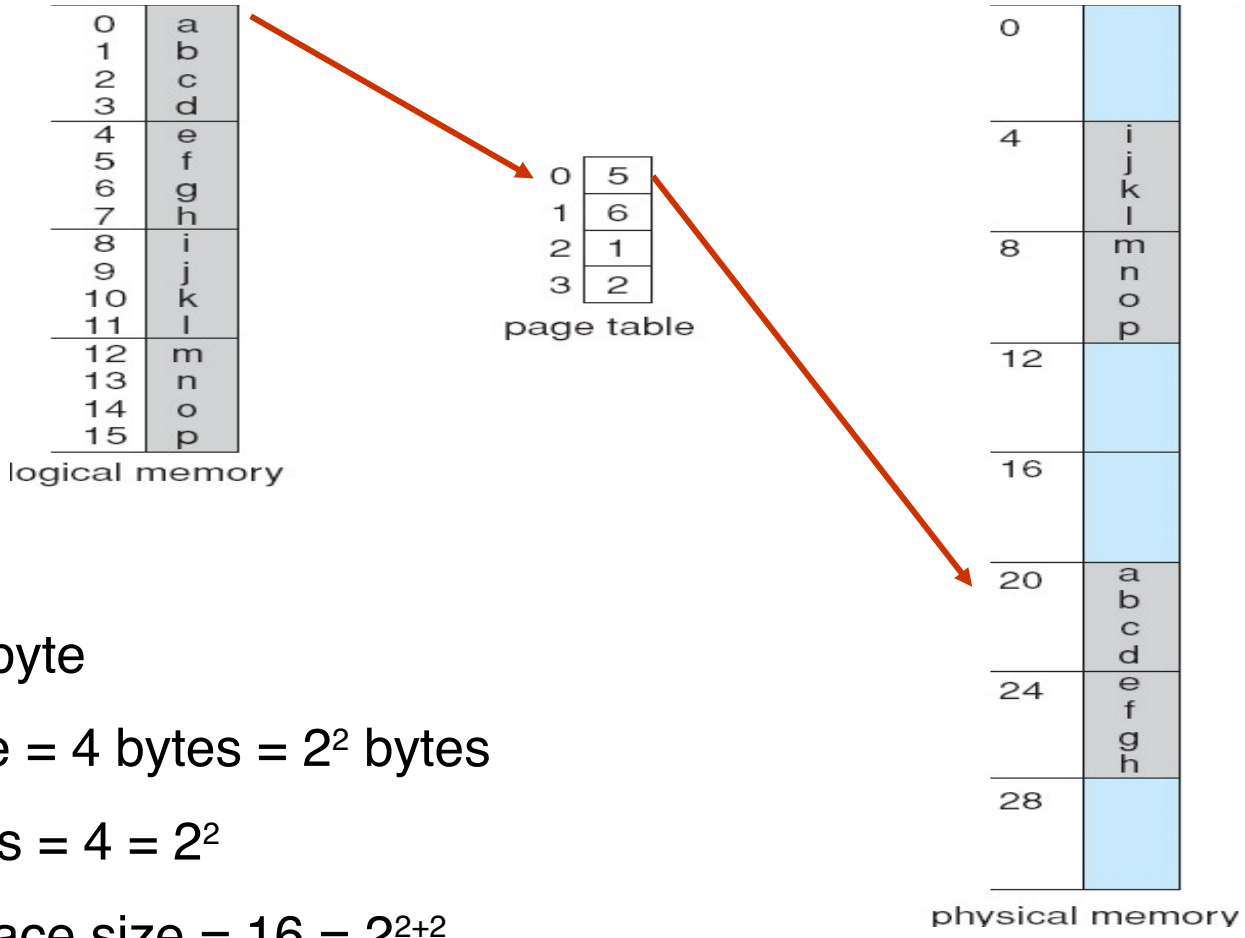
Address Translation Scheme

- Address generated by CPU is divided into:
  - ***Page number*** *(p)* –index into a ***page table*** which contains the 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

Paging: Address Translation Scheme



Paging Example



1 character = 1 byte

Frame/page size = 4 bytes = 2^2 bytes

Number of pages = 4 = 2^2

Logical addr. space size = 16 = 2^{2+2}

Physical address space size = 32 bytes

Menti question (43 75 8)

Assume:

- 32bit architecture, single level paging
- 4GB of main memory (2^32 Bytes)
- Page size of 4KB (2^12 Bytes)
- 200 running processes

What is the required size for all the page tables?

A) 200MB

B) 400MB

C) 800MB

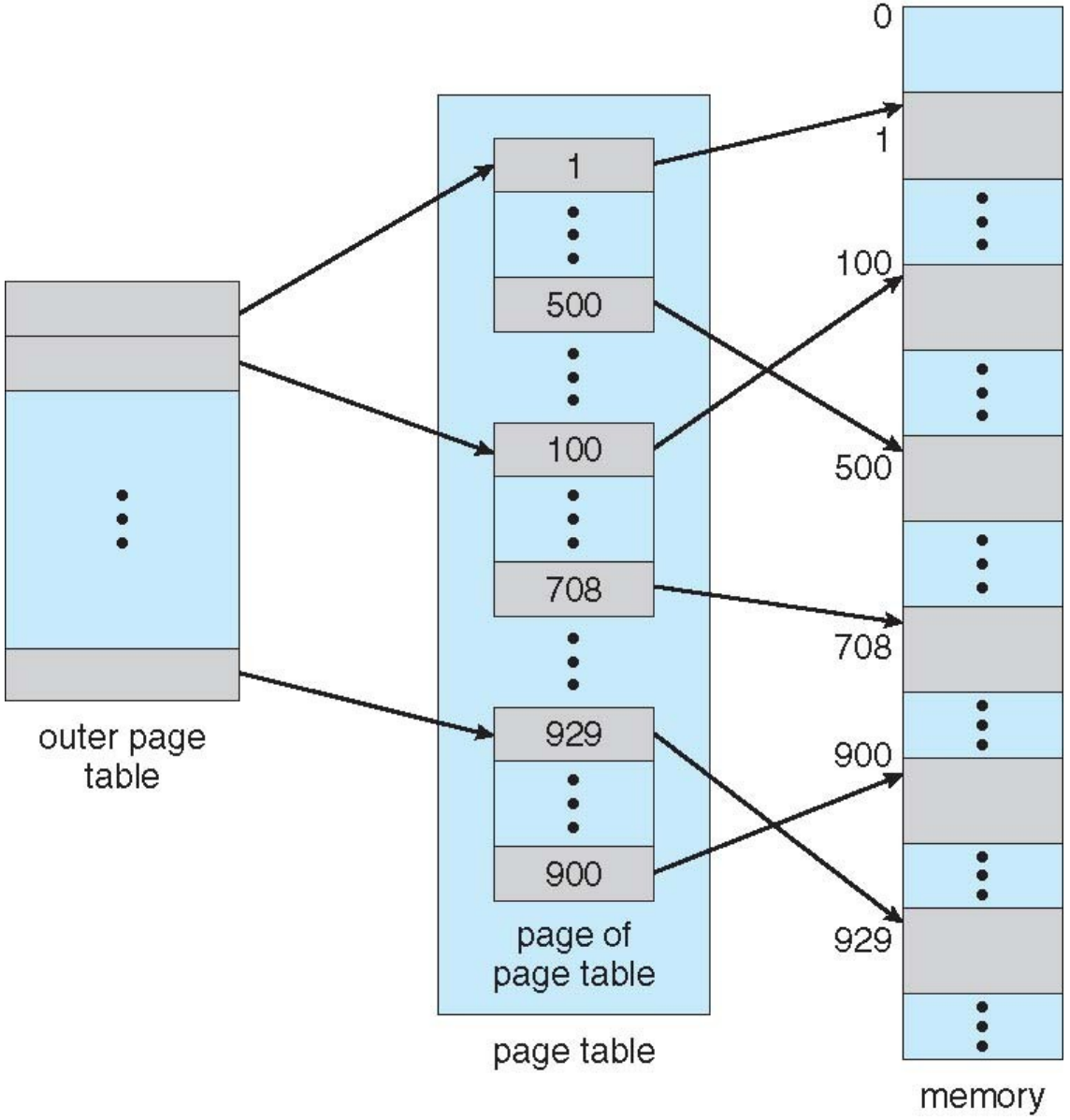
D) 1600MB

Page Table Structure

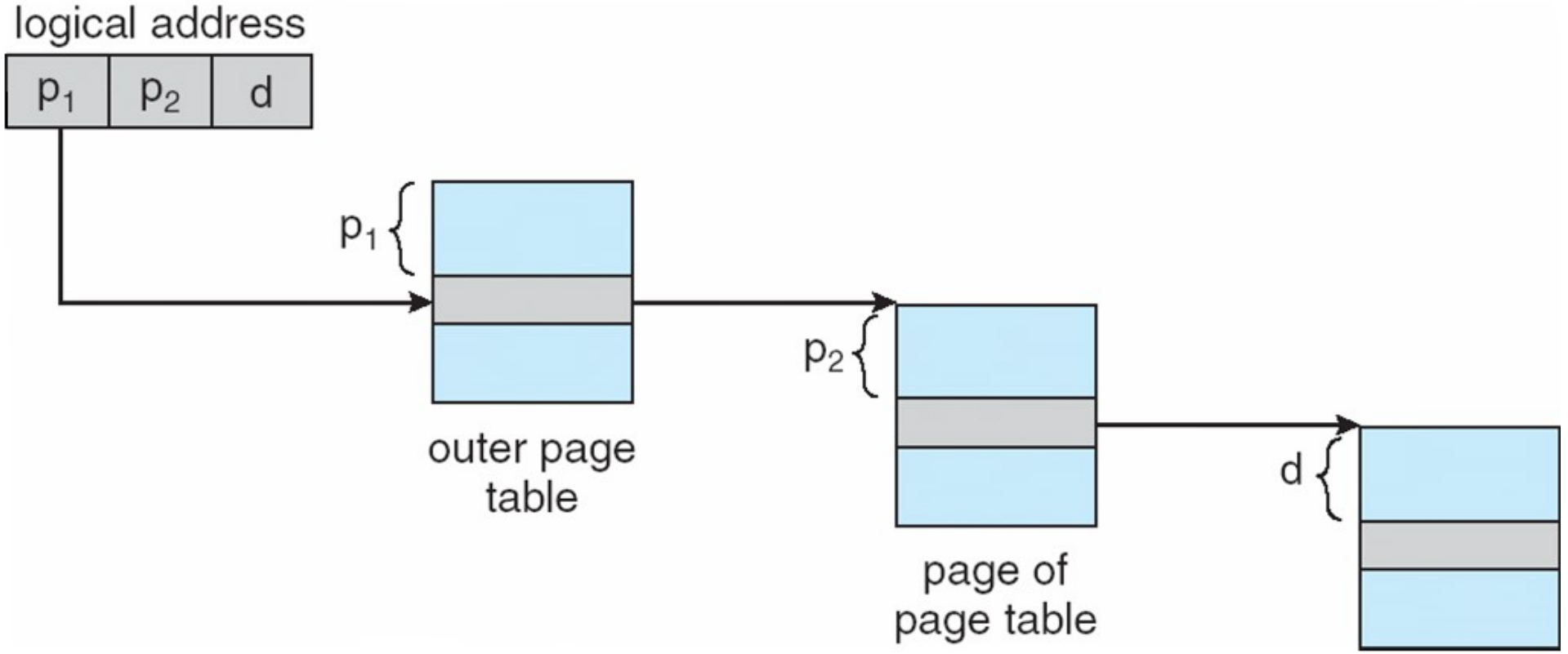
- The “large page table problem”

- Page table structures:
  - Hierarchical Paging: “page the page table”
  - Hashed Page Tables
  - Inverted Page Tables

Hierarchical Page Tables

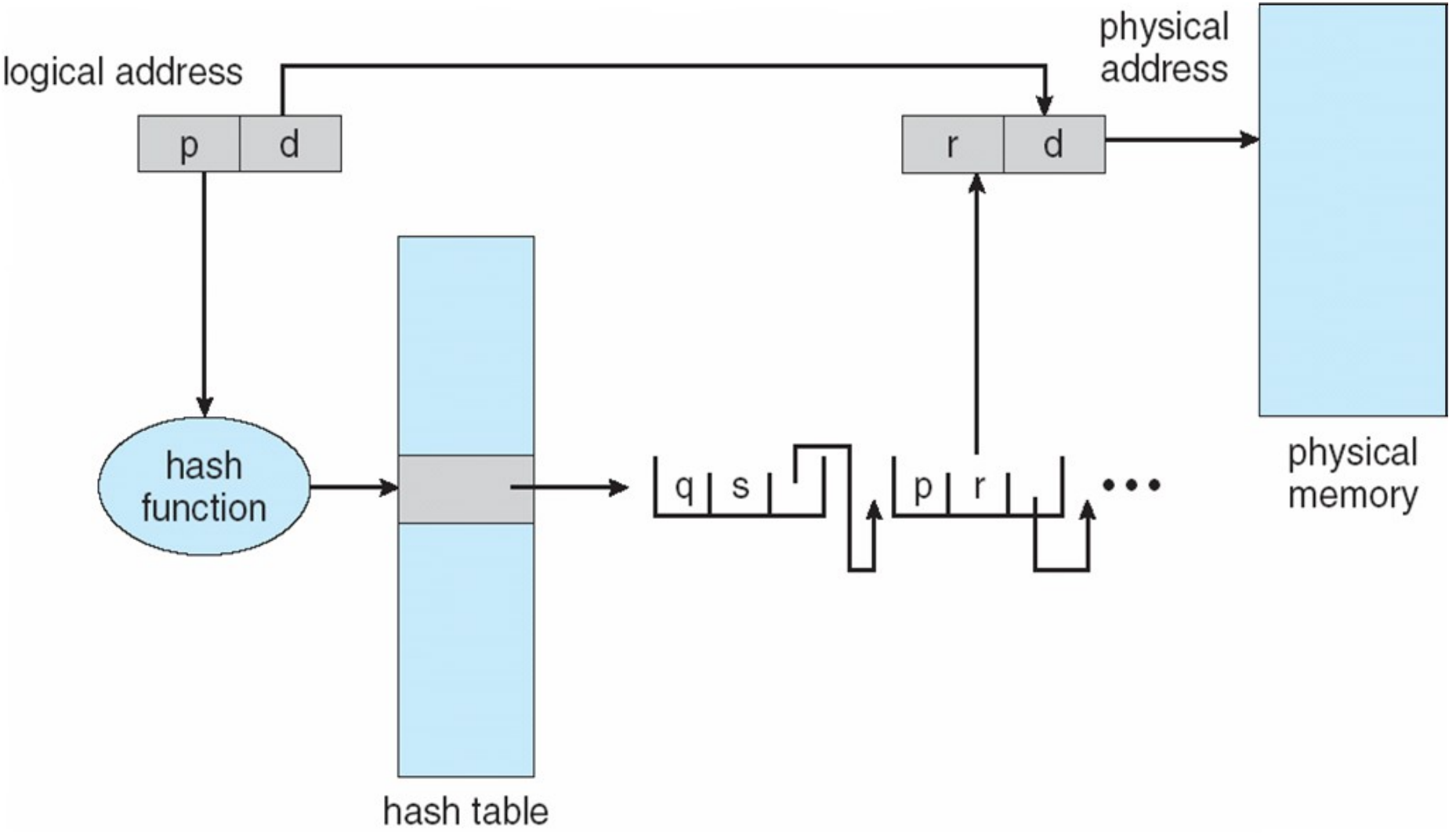


Address-Translation Scheme

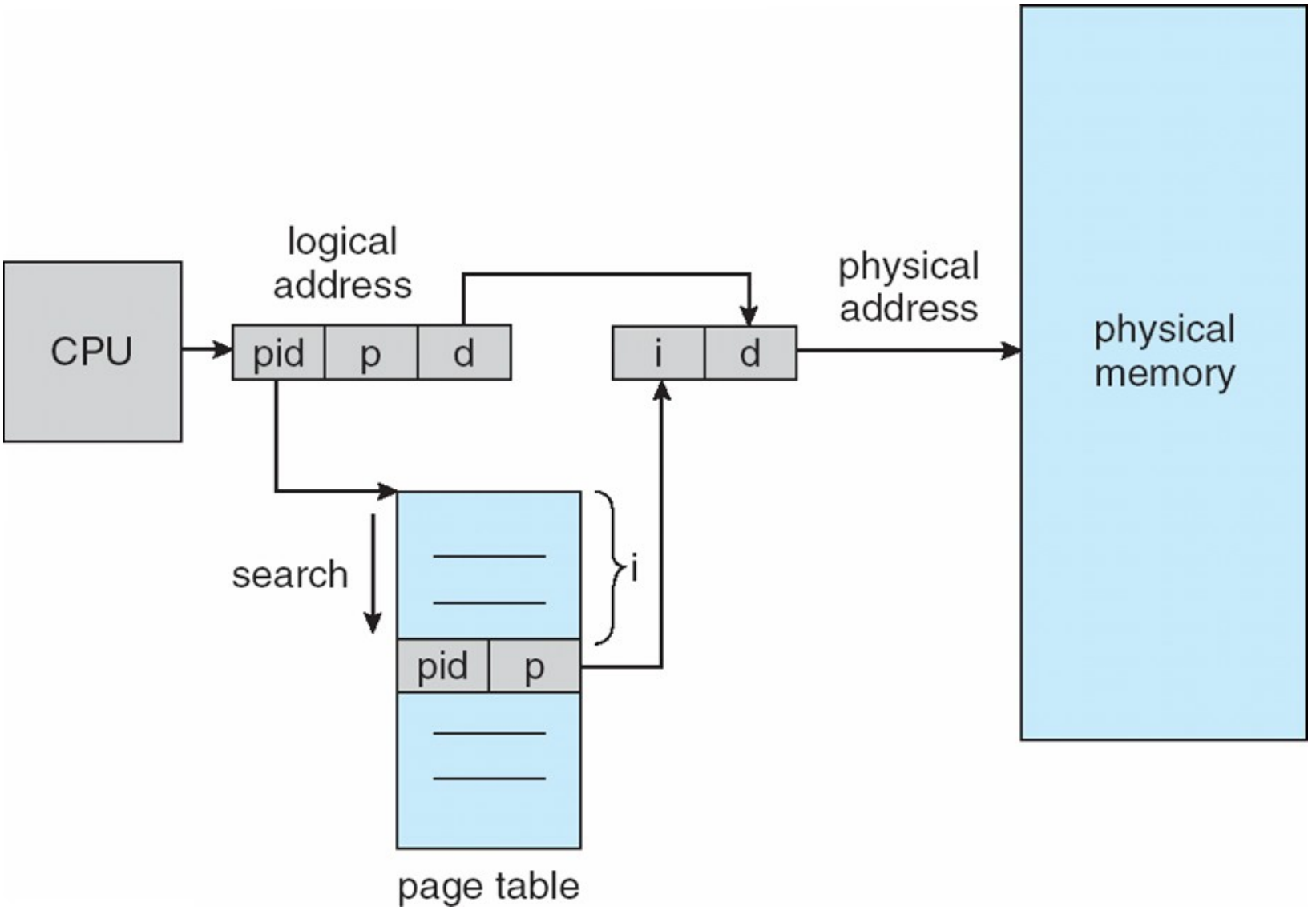


Can we address a 64-bit memory space?

Hashed Page Table



Inverted Page Table Architecture

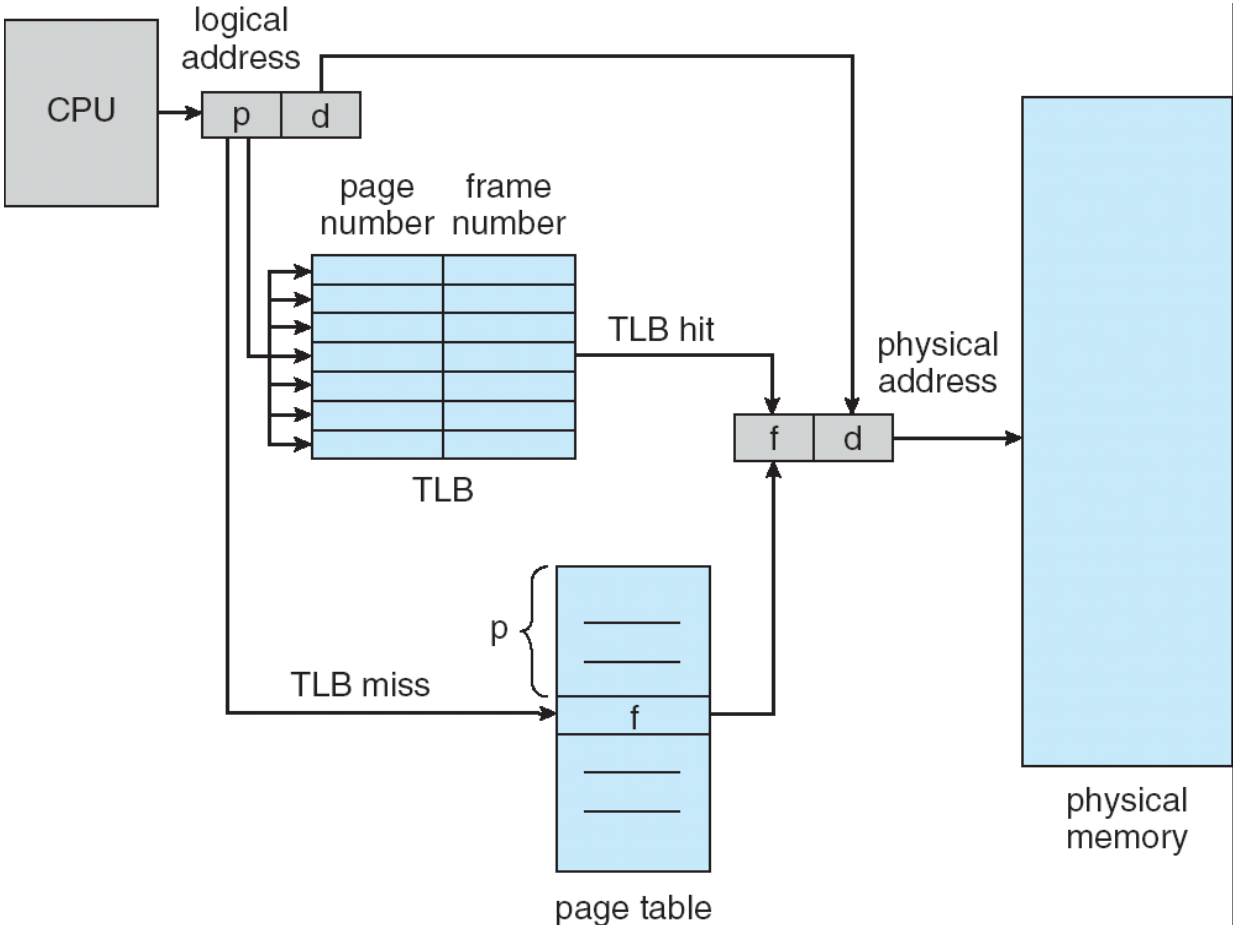


Implementation of the Page Table

- Page table is kept in main memory
- *Page-table base register (*PTBR)
  points to the page table
- *Page-table length register* (PRLR)
  indicates size of the page table
- Every data/instruction access requires n+1 memory accesses (for n-level paging).
  - One for the page table and one for the data/instruction.
- Solve the (n+1)-memory-access problem
  - by using a special fast-lookup cache (in hardware):

    translation look-aside buffer (TLB)
    - Implements an **associative memory**

Paging Hardware With TLB



TLB: fast, small, and expensive,

Typ. 64…1024 TLB entries

in main memory

Effective Access Time

- Memory cycle time: t
- Time for associative lookup: ε
- TLB hit ratio α
  - percentage of times that a page number is found in TLB

Effective Access Time

- Memory cycle time: t
- Time for associative lookup: ε
- TLB hit ratio α
  - percentage of times that a page number is found in TLB

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

$$\text{EAT} = (t + \varepsilon)\,\alpha + (2t + \varepsilon)(1 - \alpha)$$

Effective Access Time

- Memory cycle time: t
- Time for associative lookup: ε
- TLB hit ratio α
  - percentage of times that a page number is found in TLB

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

$$\text{EAT} = (t + \varepsilon)\,\alpha + (2t + \varepsilon)(1 - \alpha)$$
$$= 2t + \varepsilon - \alpha\, t$$

Effective Access Time

- Memory cycle time: t
- Time for associative lookup: ε
- TLB hit ratio α
  - percentage of times that a page number is found in TLB

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

$$\text{EAT} = (t + \varepsilon)\,\alpha + (2t + \varepsilon)(1 - \alpha)$$
$$= 2t + \varepsilon - \alpha\, t$$

Example: For t =100 ns, ε = 20 ns, α = 0.8: EAT = 140 ns

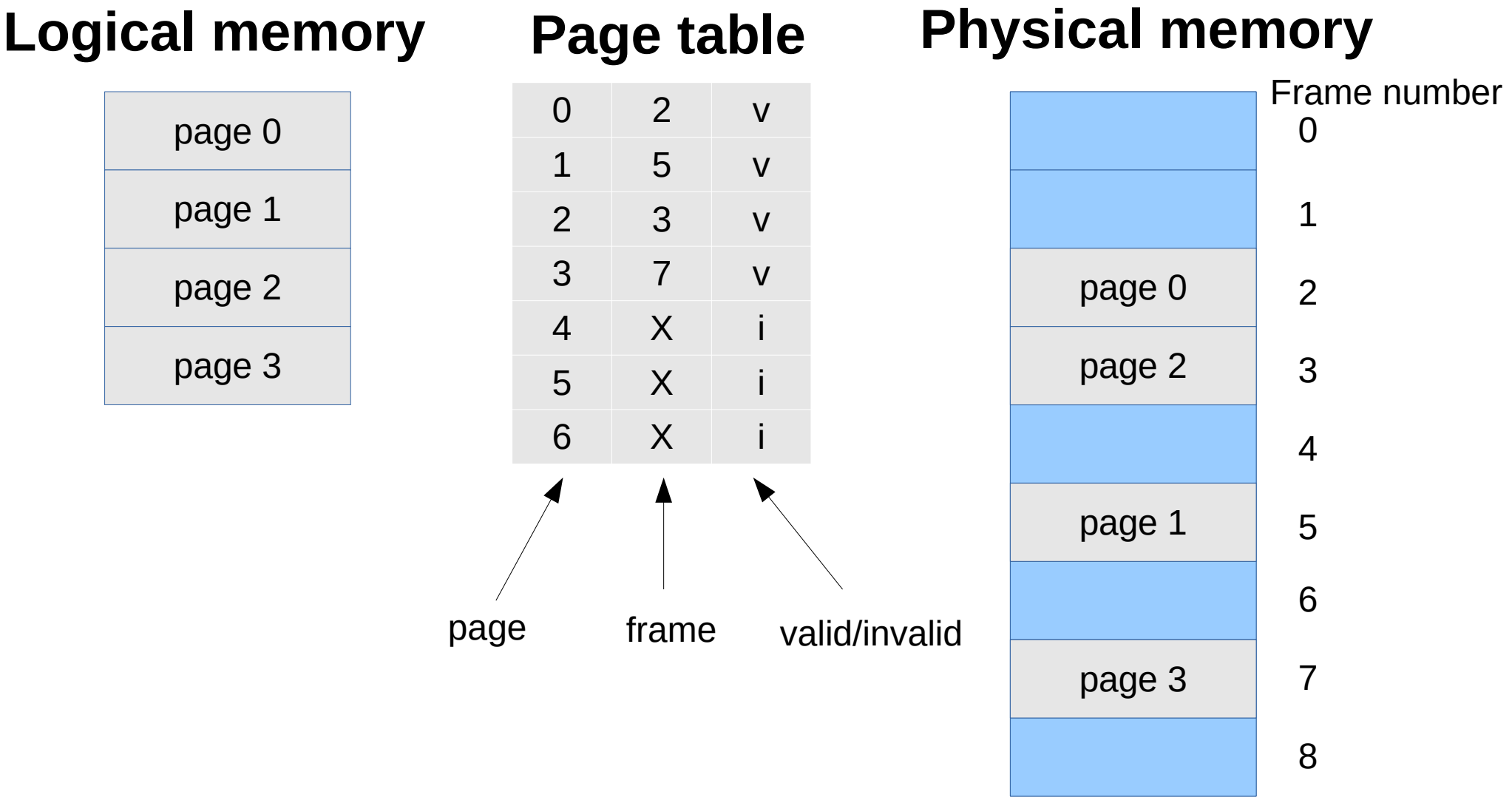
Memory Protection

- Implemented by associating protection bit with each frame

- **Valid-invalid** bit attached to each entry in the page table:
  - **“valid”**: the associated page is in the process’ logical address space, and is thus a legal page
  - **“invalid”**: the page is not in the process’ logical address space

- Allows dynamically sized page tables

Memory Protection



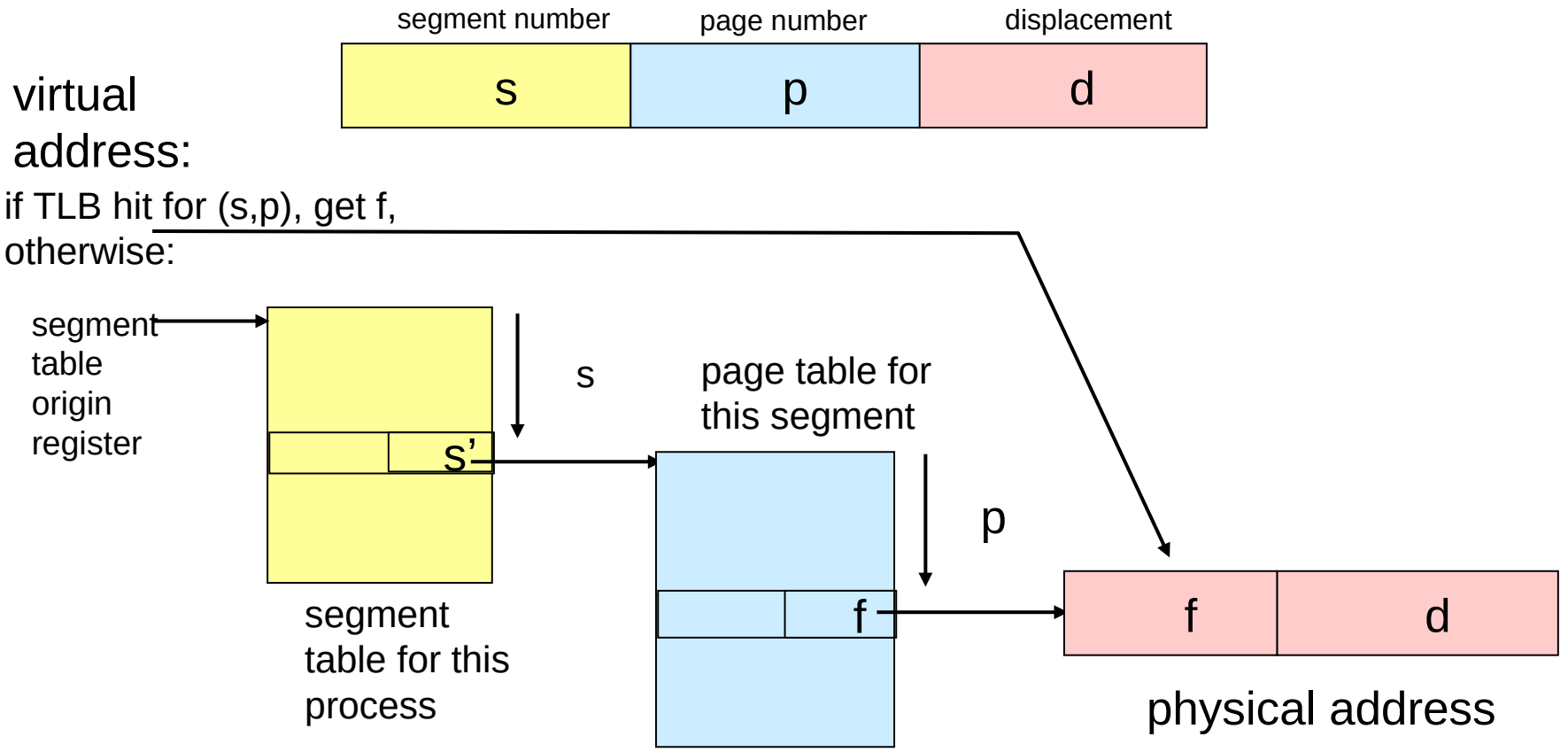
Shared memory

– Easy with paged memory!

Combining Segmentation and Paging

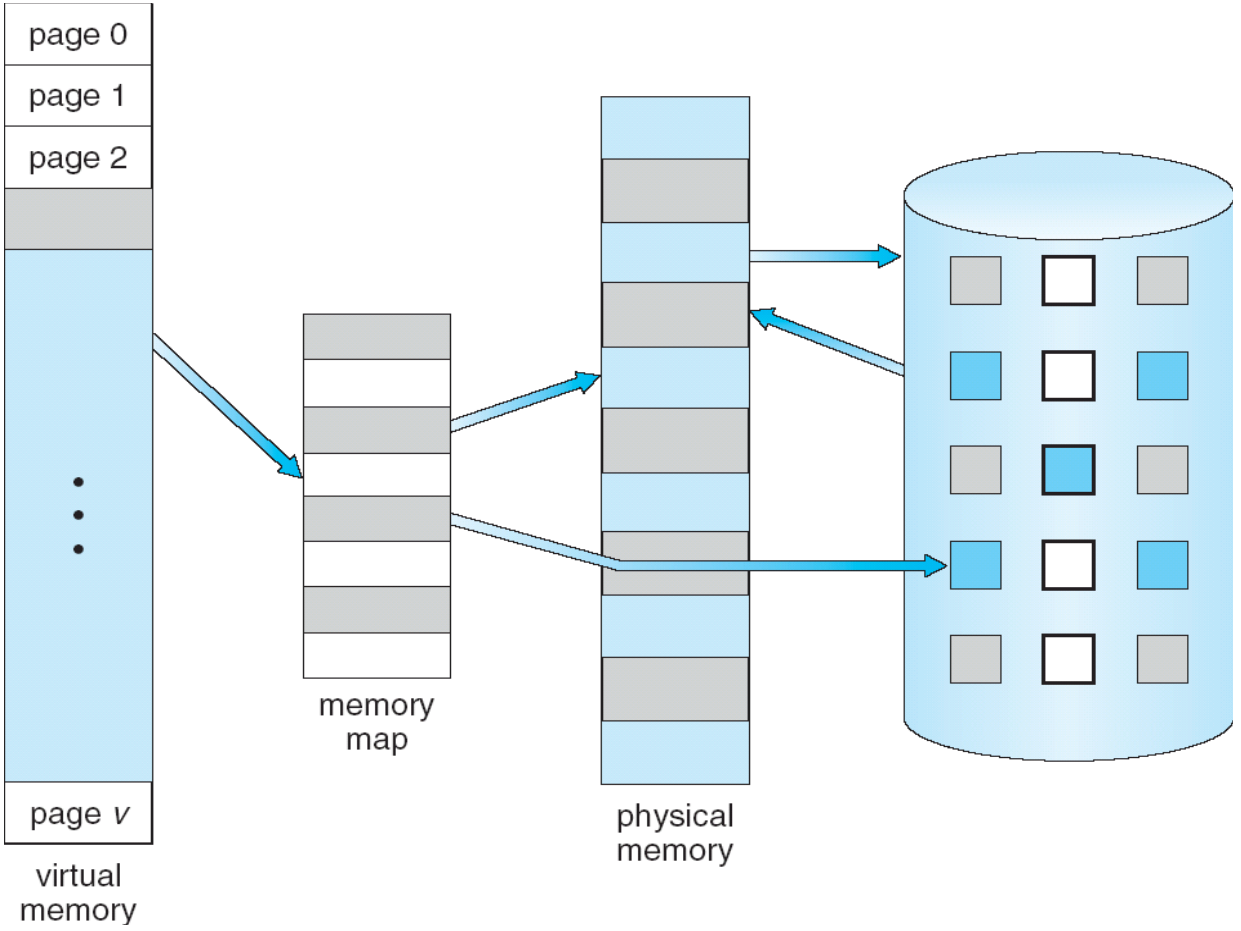
- Each segment is organized as a set of pages.
- Segment table entries refer to a page table for each segment.
- TLB used to speed up effective access time.

Combining Segmentation and Paging



Demand Paging

Virtual Memory That is Larger Than Physical Memory



[Kilburn *et al.* 1961]

Demand Paging

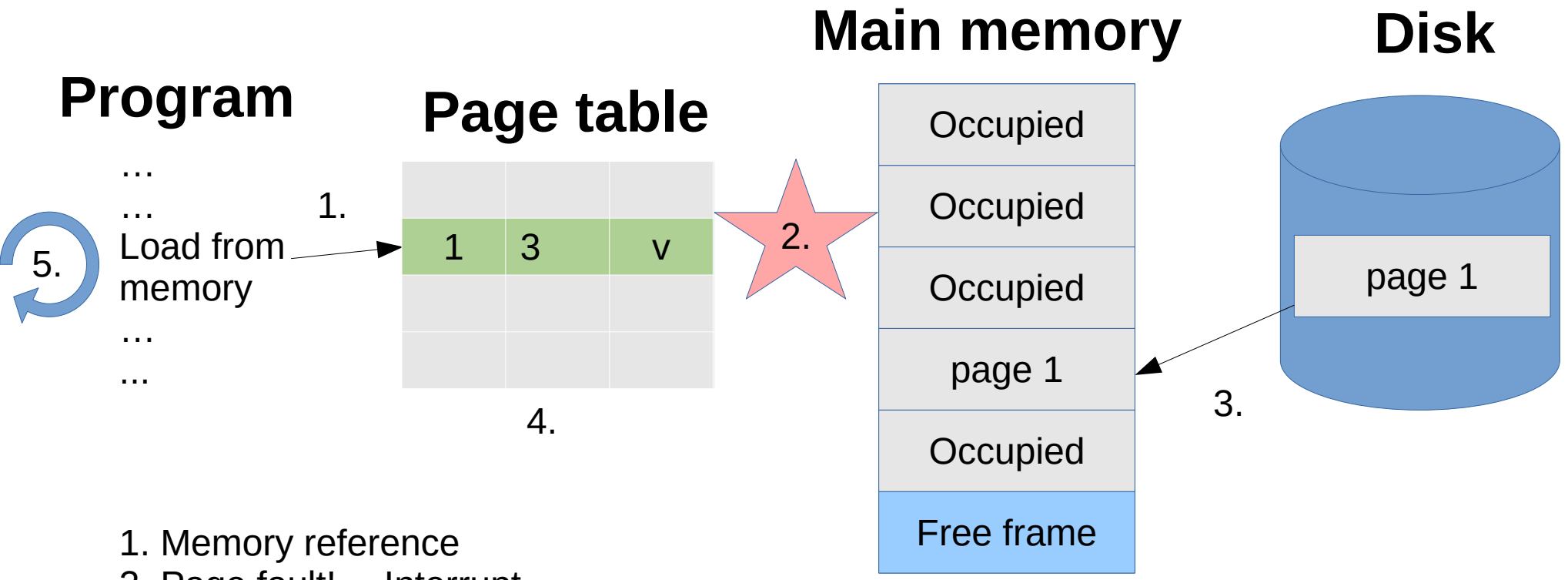
- Bring a page into memory only when it is needed
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users

- Page is needed if referenced (load/store, data/instructions)
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory

Rather than swapping entire processes (cf. *swapping*),
we *page* their pages from/to disk only when first referenced.

Steps in Handling a Page Fault

(Case: a free frame exists)



1. Memory reference
2. Page fault! → Interrupt
3. OS moves page into memory
4. Update page table
5. Restart memory access instruction

What happens if there is no free frame?

Page replacement

- Find some page in memory, but not really in use, swap it out
  - Write-back only necessary if victim page was modified
  - Same page may be brought into memory several times

- More details next lecture...

Menti question (43 75 8)

Which of the following memory management tasks can be performed by the MMU:

A) Memory protection

B) Page table lookup

C) Page replacement

D) TLB lookup

Performance of Demand Paging

Performance of Demand Paging

- ***Page Fault Rate*** p $0 \le p \le 1.0$
  - if $p = 0$: no page faults
  - if $p = 1$, every reference is a fault
- ***Write-back rate*** w $0 <= w <= 1$
- Memory access time t

Performance of Demand Paging

- ***Page Fault Rate*** p $0 \le p \le 1.0$
  - if $p = 0$: no page faults
  - if $p = 1$, every reference is a fault
- ***Write-back rate*** w $0 <= w <= 1$
- Memory access time t

- **Effective Access Time** (EAT)

Performance of Demand Paging

- ***Page Fault Rate*** p $0 \le p \le 1.0$
  - if $p = 0$: no page faults
  - if $p = 1$, every reference is a fault
- ***Write-back rate*** w $0 <= w <= 1$
- Memory access time t

- **Effective Access Time** (EAT)

  EAT = $(1 - p)\ t$ +

Performance of Demand Paging

- ***Page Fault Rate*** p $0 \le p \le 1.0$
  - if $p = 0$: no page faults
  - if $p = 1$, every reference is a fault
- ***Write-back rate*** w $0 <= w <= 1$
- Memory access time t

- **Effective Access Time** (EAT)

  EAT = $(1 - p)\ t + p$ (

Performance of Demand Paging

- ***Page Fault Rate*** p $0 \le p \le 1.0$
  - if $p = 0$: no page faults
  - if $p = 1$, every reference is a fault
- ***Write-back rate*** w $0 <= w <= 1$
- Memory access time t

- **Effective Access Time** (EAT)

  EAT = $(1 - p)\ t + p$ (page fault overhead

Performance of Demand Paging

- ***Page Fault Rate*** p $0 \le p \le 1.0$
  - if $p = 0$: no page faults
  - if $p = 1$, every reference is a fault
- ***Write-back rate*** w $0 <= w <= 1$
- Memory access time t

- **Effective Access Time** (EAT)

$$EAT = (1 - p)\, t + p \; (\text{page fault overhead} + w \; (\text{time to swap page out})$$

Performance of Demand Paging

- ***Page Fault Rate*** p $0 \le p \le 1.0$
  - if $p = 0$: no page faults
  - if $p = 1$, every reference is a fault
- ***Write-back rate*** w $0 <= w <= 1$
- Memory access time t

- **Effective Access Time** (EAT)

$$EAT = (1 - p)\, t + p \; (\text{page fault overhead} + w\,(\text{time to swap page out}) + \text{time to swap new page in})$$

Performance of Demand Paging

- ***Page Fault Rate*** p $0 \le p \le 1.0$
  - if $p = 0$: no page faults
  - if $p = 1$, every reference is a fault
- ***Write-back rate*** w $0 <= w <= 1$
- Memory access time t

- **Effective Access Time** (EAT)

$$EAT = (1 - p)\, t + p \; (\text{page fault overhead} + w\,(\text{time to swap page out}) + \text{time to swap new page in} + \text{restart overhead})$$

Performance of Demand Paging

- ***Page Fault Rate*** p $0 \le p \le 1.0$
  - if $p = 0$: no page faults
  - if $p = 1$, every reference is a fault
- ***Write-back rate*** w $0 <= w <= 1$
- Memory access time t

- **Effective Access Time** (EAT)

$$EAT = (1 - p)\, t + p \; (\text{page fault overhead} + w \; (\text{time to swap page out}) + \text{time to swap new page in} + \text{restart overhead} + t \;)$$