Topics: Memory Management
(SGG, Chapter 08)
8.1, 8.2, 8.3, 8.5, 8.6

CS 3733 Operating Systems

Instructor: Dr. Turgay Korkmaz

Department Computer Science

The University of Texas at San Antonio

Office: NPB 3.330
Phone: (210) 458-7346
Fax: (210) 458-4437
e-mail: korkmaz@cs.utsa.edu
web: www.cs.utsa.edu/~korkmaz

These slides are prepared based on the materials provided by the textbook [**SGG**] .

Objectives

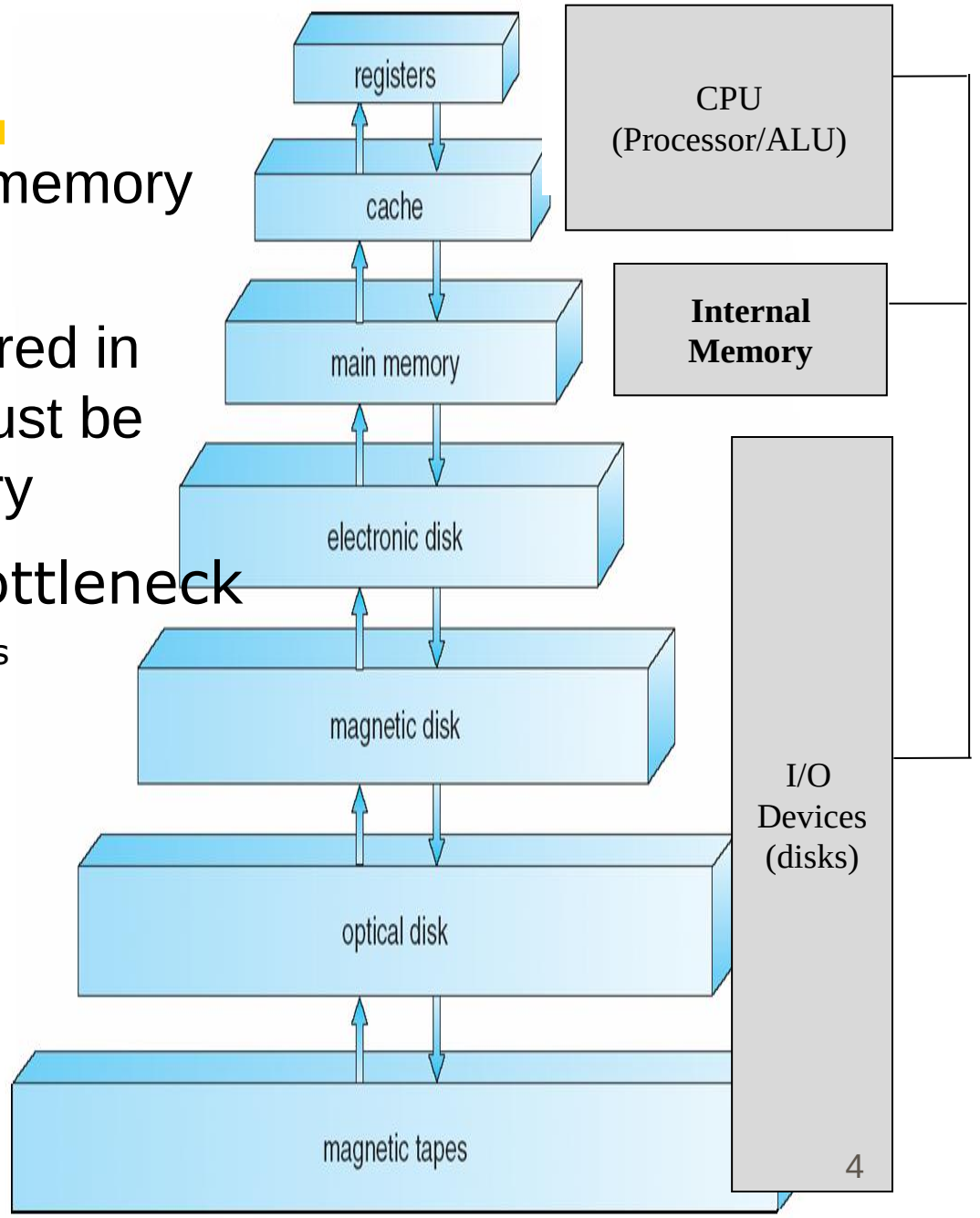
- To provide a detailed description of various ways of organizing memory hardware
- To learn how to share memory among multiple processes using various memory-management techniques , including **paging** and **segmentation**
- To provide a detailed description of the Intel Pentium, which supports both pure segmentation and segmentation with paging

Outline

- Background
  - Logical vs. physical addresses
  - Address binding
- Partitions: base + limit registers
- Swapping
- Contiguous Memory Allocation and Fragmentation
- **Paging**
- **Translation Lookaside Buffer (TLB)**
  - **Effective memory access time**
- Other Page Table Structures
- Pentium and Linux page tables

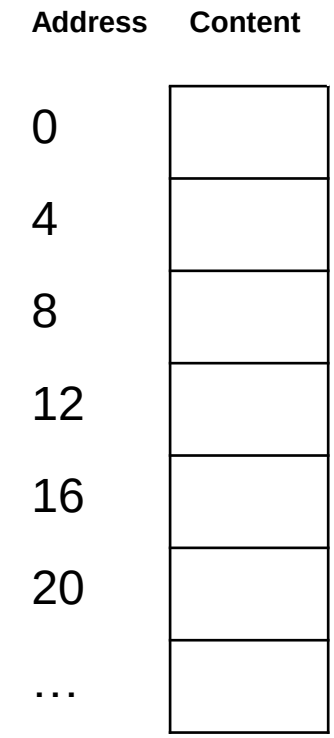
Background

- CPU can directly access main memory and registers only
- But, programs and data are stored in disks. So, program and data must be brought (from disk) into memory
- Memory accesses can be bottleneck
  - **Cache** between memory and CPU registers
- Memory Hierarchy
  - Cache: small, fast, expensive; SRAM
  - **Main memory**: medium-speed; DRAM
  - Disk: slow, cheap, non-volatile storage



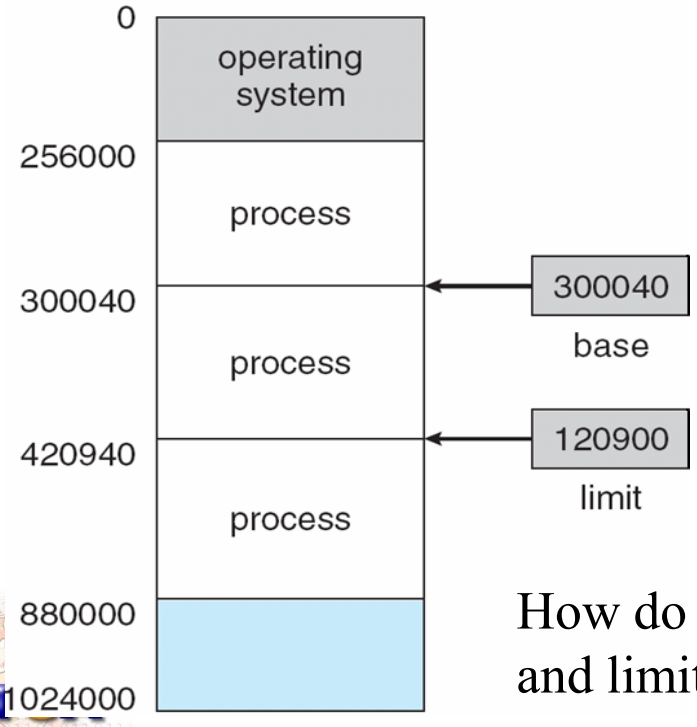
Background

- Think memory as an array of words containing program instructions and data
- How do we execute a program?
  - **Fetch an instruction → decode → may fetch operands → execute → may store results**
- Memory unit sees a stream of **ADDRESSES**
- How to manage and protect main memory while sharing it among multiple processes?
  - Keeping multiple process in memory is essential to improving the CPU utilization

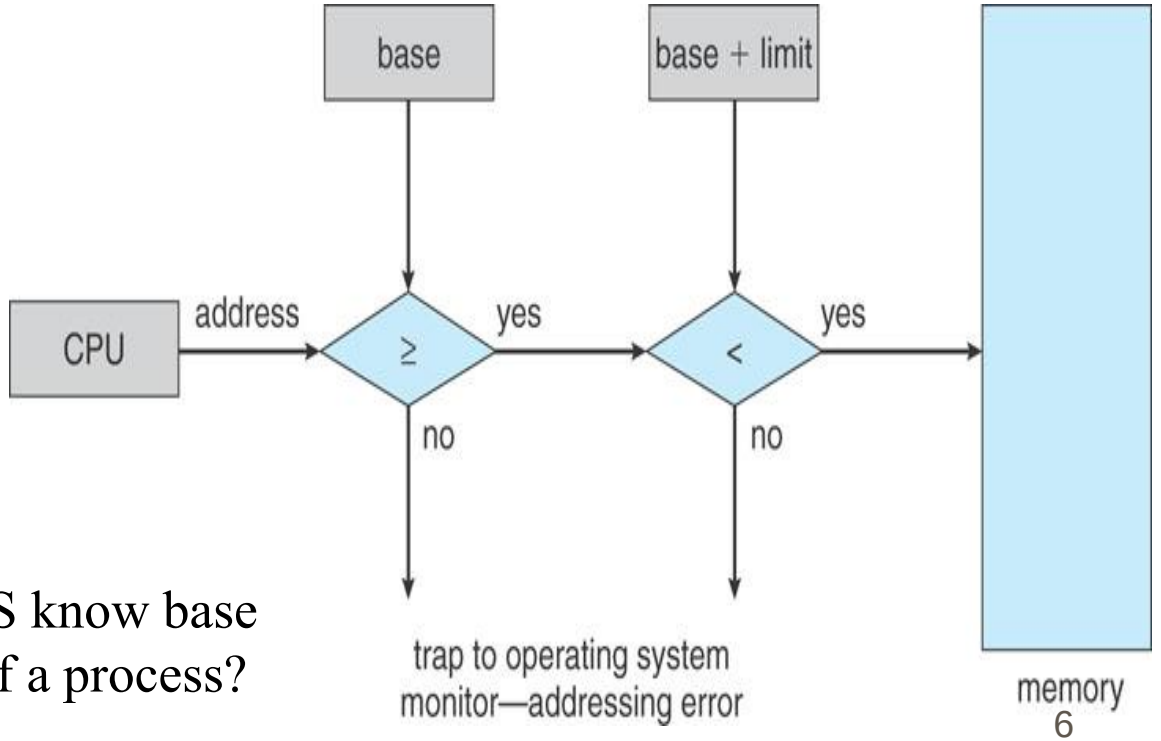


Simple one: Base and Limit Registers

- Memory protection is required to ensure correct operation
- A pair of **base** and **limit** registers define the logical address space of a process. Every memory access is checked by hardware.



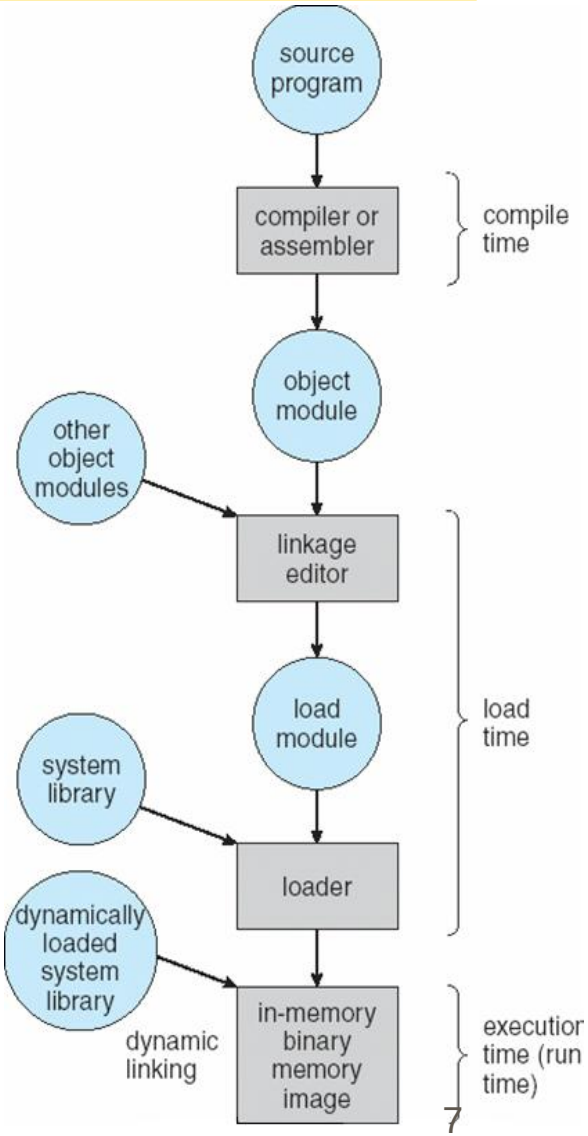
How do OS know base and limit of a process?



How to Bind Instructions and Data to Memory

Mapping from one address space to the other

- Address binding of instructions and data to memory addresses can happen at three different stages
  - **Compile time**: If memory location is known beforehand, **absolute code** can be generated; must recompile code if starting location changes (e.g. DOS .com programs)
  - **Load time**: Compiler generates **relocatable code**, final binding occurred at load time
  - **Execution time**: Binding delayed until execution time if the process can be moved during its execution from one memory segment to another



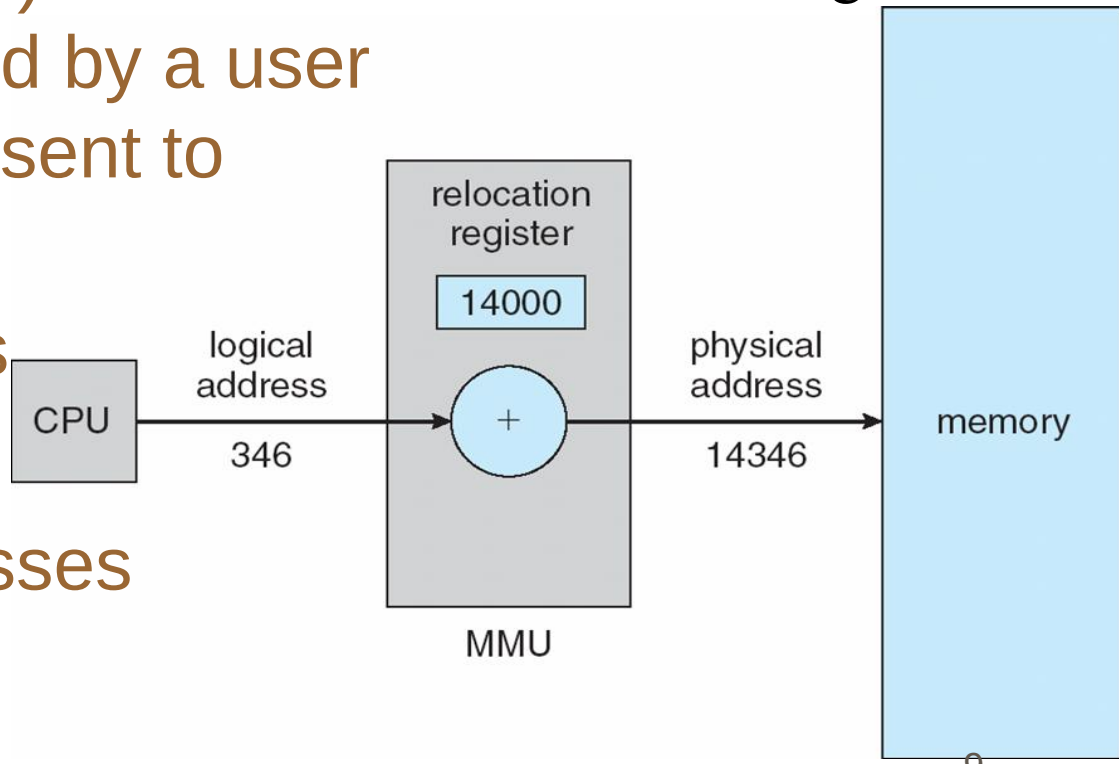
Logical vs. Physical Address Space

- **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
  - The mapping form logical address to physical address is done by a hardware called a *memory management unit* (MMU).
  - We will mainly study how this mapping is done and what hardware support is needed

Memory-Management Unit (MMU)

- Hardware device that maps logical (virtual) address to physical address
- In a simple MMU, the value in the relocation register (base) 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, not *real* physical addresses

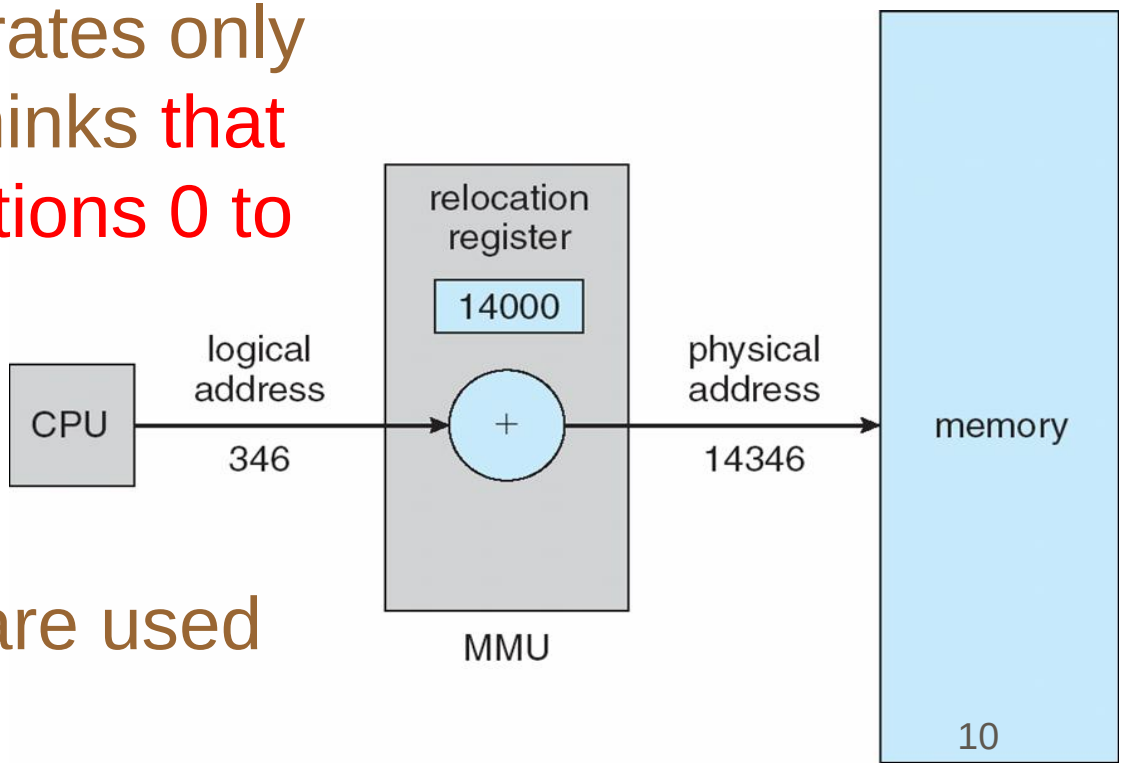
Dynamic relocation using a relocation register



A simple MMU Example

- logical addresses (in the range 0~max) and physical addresses (in the range R+0 to R+max for a base value R).
- The user program generates only logical addresses and thinks that the process runs in locations 0 to max.
- logical addresses must be mapped to physical addresses before they are used

Dynamic relocation using a relocation register



Partitions for Memory Management

- **Fixed** Partitions
  - Divide memory into **fixed size** of partitions (**not necessarily equal**)
  - Each partition for at most one process
  - Base + limit registers for relocation and protection
  - How to determine the partition sizes?
- **Variable** Partitions
  - Partition sizes determined dynamically
  - OS keeps a table of current partitions
  - When a job finishes, leaves a partition hole
  - Consolidate free partitions → **compaction**

Dynamic Loading

- Just load main routine
- Dynamically load other routines when they are called
- Why dynamic loading?
  - Without this, the size of a process is limited to that of physical memory
  - Better memory-space utilization; unused routine is never loaded
  - Useful when large amounts of code are needed to handle infrequently occurring cases *(error handling)*
- No special support from the operating system is required, implemented through program design
- OS may provide library functions

Dynamic Linking and Shared Libraries

- Static linking
  - System language and library routines are included in the binary code
- Dynamic linking: similar to dynamic loading
  - Linking postponed until execution time
  - Without that, every library will have a copy in the executable file, wasting both disk space and memory
  - A *stub* is used to locate and load the appropriate memory-resident library routine, when the routine is not existing
  - If it is existing, no need for loading again (PLT)
- Dynamic linking is particularly useful for libraries

  (one copy, transparent updates)
- Requires support form OS

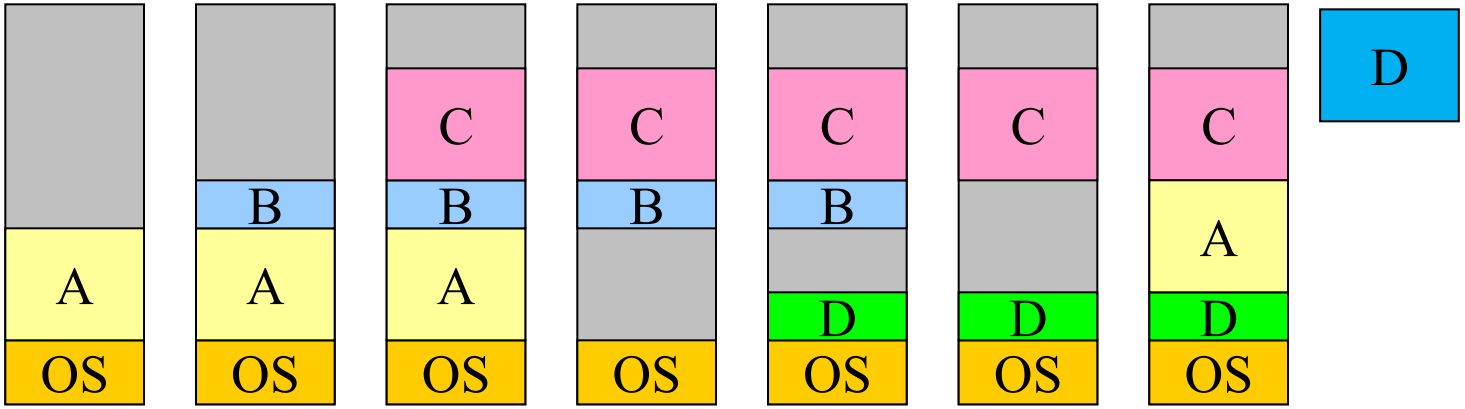
Memory Management

- Background
- Swapping
- Contiguous Memory Allocation and Fragmentation
- Paging
- Page Table implementation, TLB, access time
- Other Page Table Implementations and Structures

Swapping

Consider a multi-programming environment:

- Each program must be in the memory to be executed
- Processes come into memory and
- Leave memory when execution is completed



Reject it! But if you want to support more processes,

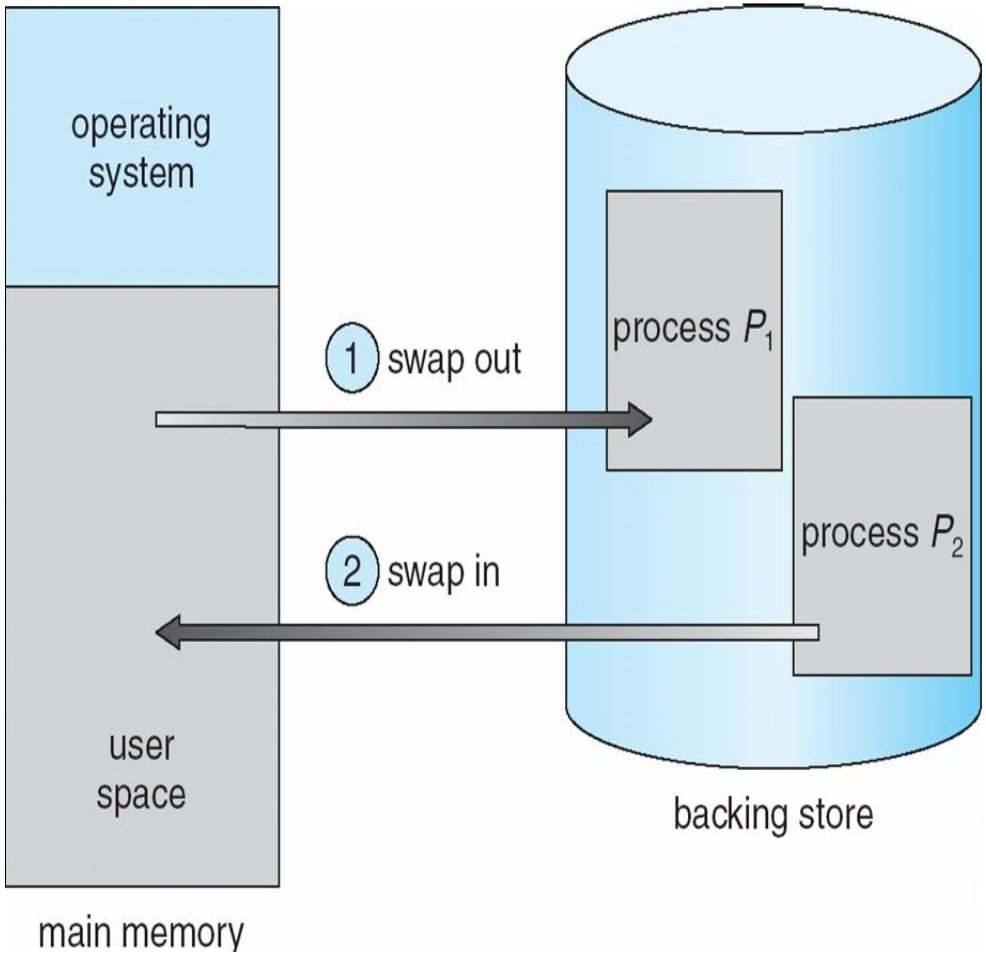
What if no free region is big enough?

Swap out an old process to a disk

(waiting for long I/O, quantum expired etc)

Swapping

- A process can be swapped temporarily out of memory to a **backing store**, and then brought back into memory for continued execution
  - **Backing store** – 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
- Swapping can free up memory for additional processes.



Swapping (cont'd)

- Major part of swap time is **transfer time**;
  - Total transfer time is directly proportional to the amount of memory swapped (e.g., 10MB process / 40MB per sec = 0.25 sec)
  - May take too much time to be used often
- Standard swapping requires too much swapping time
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows), but it is **often disabled**

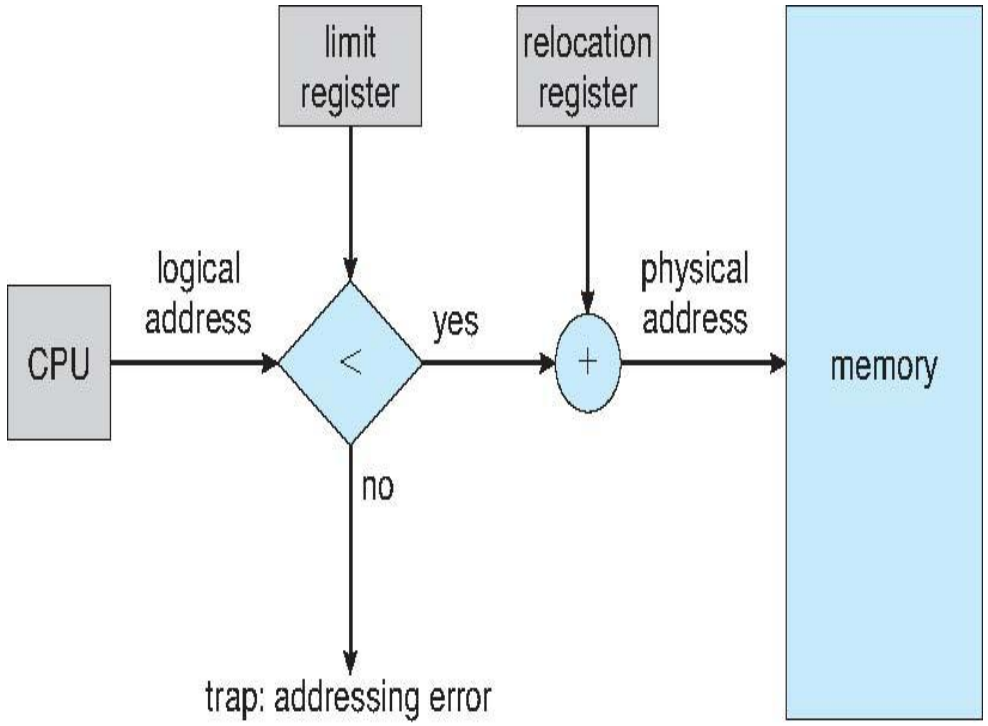
Memory Management

- Background
- Swapping
- Contiguous Memory Allocation and Fragmentation
- Paging
- Page Table implementation, TLB, access time
- Other Page Table Implementations and Structures

Contiguous Allocation

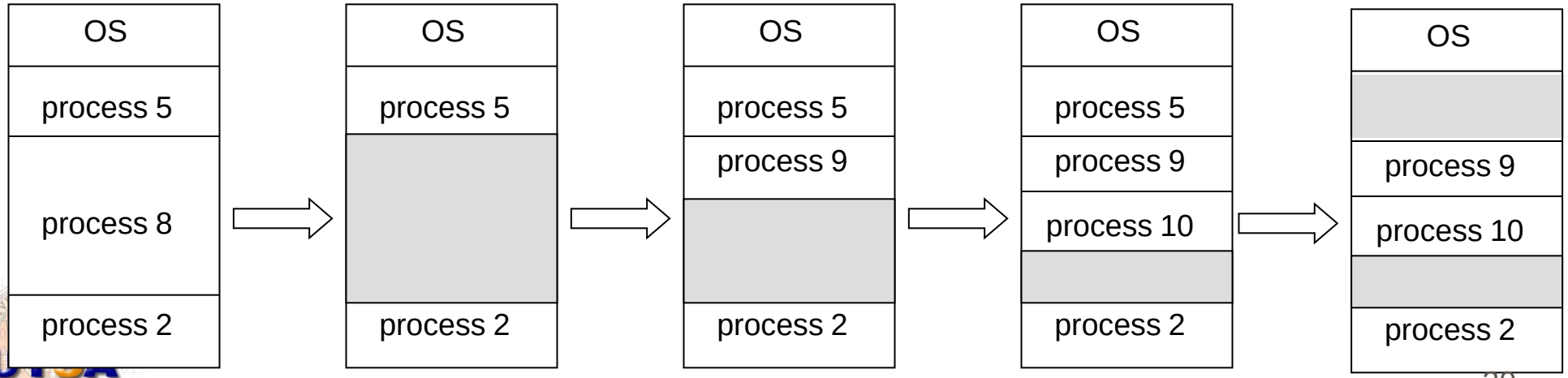
- Main memory is usually divided into two partitions:
  - Resident operating system, usually held in low memory
  - User processes, usually held in high memory

- Relocation registers are used to protect user processes from each other, and from changing operating-system code and data
  - MMU maps logical addresses to physical addresses *dynamically*
  - **But the physical addresses should be contiguous**



Contiguous Allocation (Cont)

- Multiple-partition allocation
  - **Hole** – block of available memory;
    - ✓ holes of various size are scattered throughout memory
  - When a process arrives, OS allocates memory from a hole large enough to accommodate it
  - Operating system maintains information about:
    a) allocated partitions b) free partitions (holes)



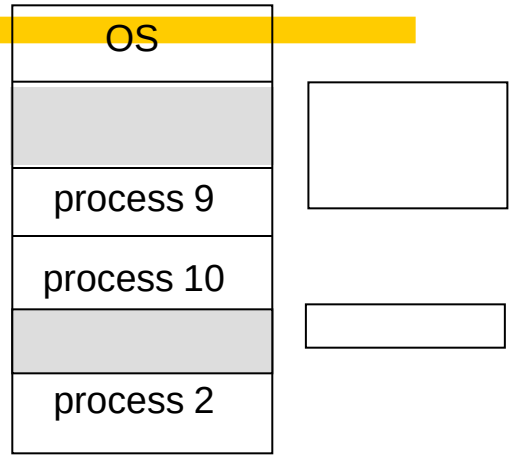
Dynamic Storage-Allocation Problem

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 are better than worst-fit in terms of speed and storage utilization. But all suffer from **fragmentation**

Fragmentation

- **External Fragmentation**
  - total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation**
  - allocated memory may be slightly larger than requested memory;
  - this size difference is internal fragmentation
- How can we reduce external fragmentation
  - **Compaction:** Move memory to place all free memory together in one large block,
    - ✓ possible *only* if relocation is dynamic and is done at execution time
    - ✓ I/O problem → large overhead



How about **not** requiring programs to be loaded **contiguously?**

Exercise:

- Suppose we use a linked list to keep track of free holes

```
struct item {
    int size;
    void *base;
    struct item *next;
}
struct item *HEAD;
```

How about implementing

```
void
perf_compaction();
```

Implement the following functions:

```
struct item *get_first_fit(int demand);
struct item *get_best_fit(int demand);
struct item *get_worst_fit(int demand);
void insert(int size, void *base);
```

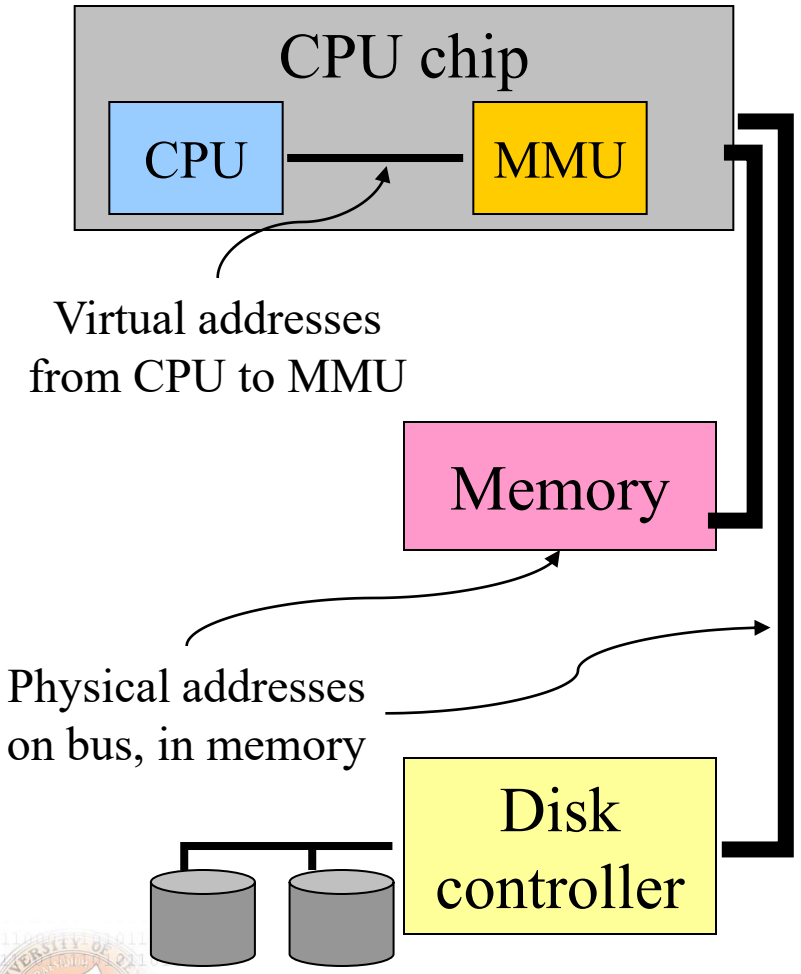
Memory Management

- Background
- Swapping
- Contiguous Memory Allocation and Fragmentation
- Paging

Not requiring programs to be loaded **contiguously?**

- Page Table implementation, TLB, access time
- Other Page Table Implementations and Structures

Logical/Virtual and Physical Addresses

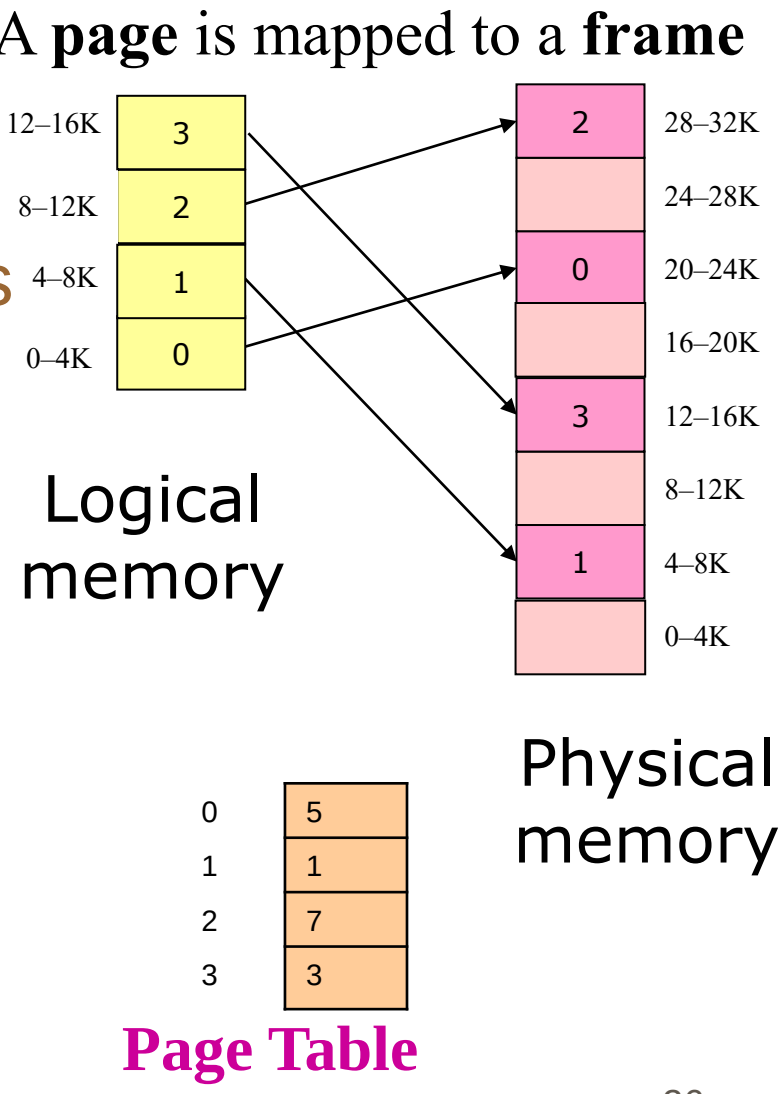


- Virtual address space
  - Determined by instruction width
  - Same for all processes
- Physical memory indexed by physical addresses
  - Limited by bus size (# of bits)
  - Amount of available memory

Paging: a memory-management scheme that permits address space of process to be non-continuous.

Paging: Basic Ideas

- Divide physical memory into fixed-sized blocks called **frames**
  - Size is power of 2, between 512 bytes and 16MB or more
- Divide logical memory into blocks of same size called **pages**
- To run a program with size ***n*** pages, we need ***n*** free frames
- Set up a **page table** to translate logical to physical addresses
  - User sees memory a logically contiguous space (0 to MAX) but OS does not need to allocate it this way



Page Table

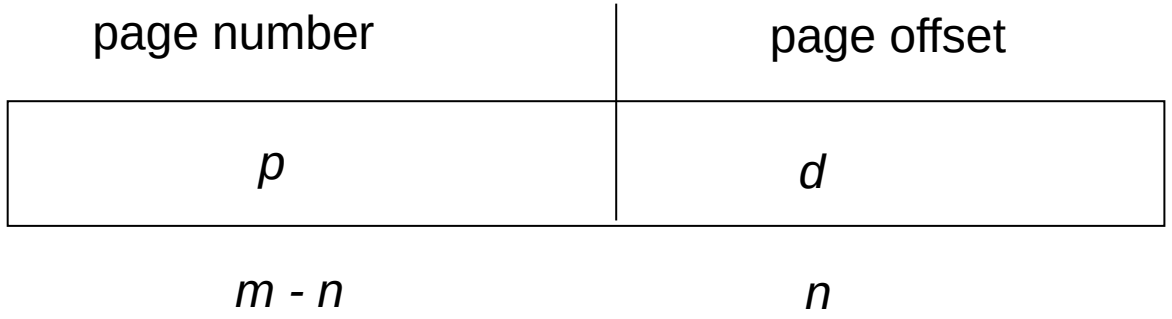
$2^{44} = 2^4 \; 2^{10} \; 2^{10} \; 2^{10} \; 2^{10}$
16 K M G T

Address Translation

- How many **pages** do we have?
- Suppose the logical address space is 2^m *and page size is* 2^n
  - Then, the number of pages is $2^m / 2^n$, which is 2^{m-n}

Logical Address (*m* bits) is divided into:

- **Page number (*p*)** – used as an index into a *page table* which contains the corresponding frame number in physical memory
- **Page offset (d)** – combined with the frame number to define the physical memory address that is sent to the memory unit



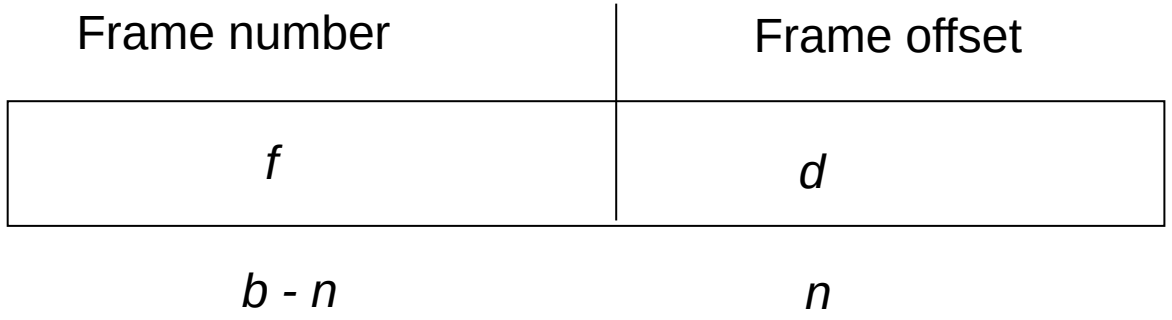
$2^{44} = 2^4 \; 2^{10} \; 2^{10} \; 2^{10} \; 2^{10}$
16 K M G T

Address Translation

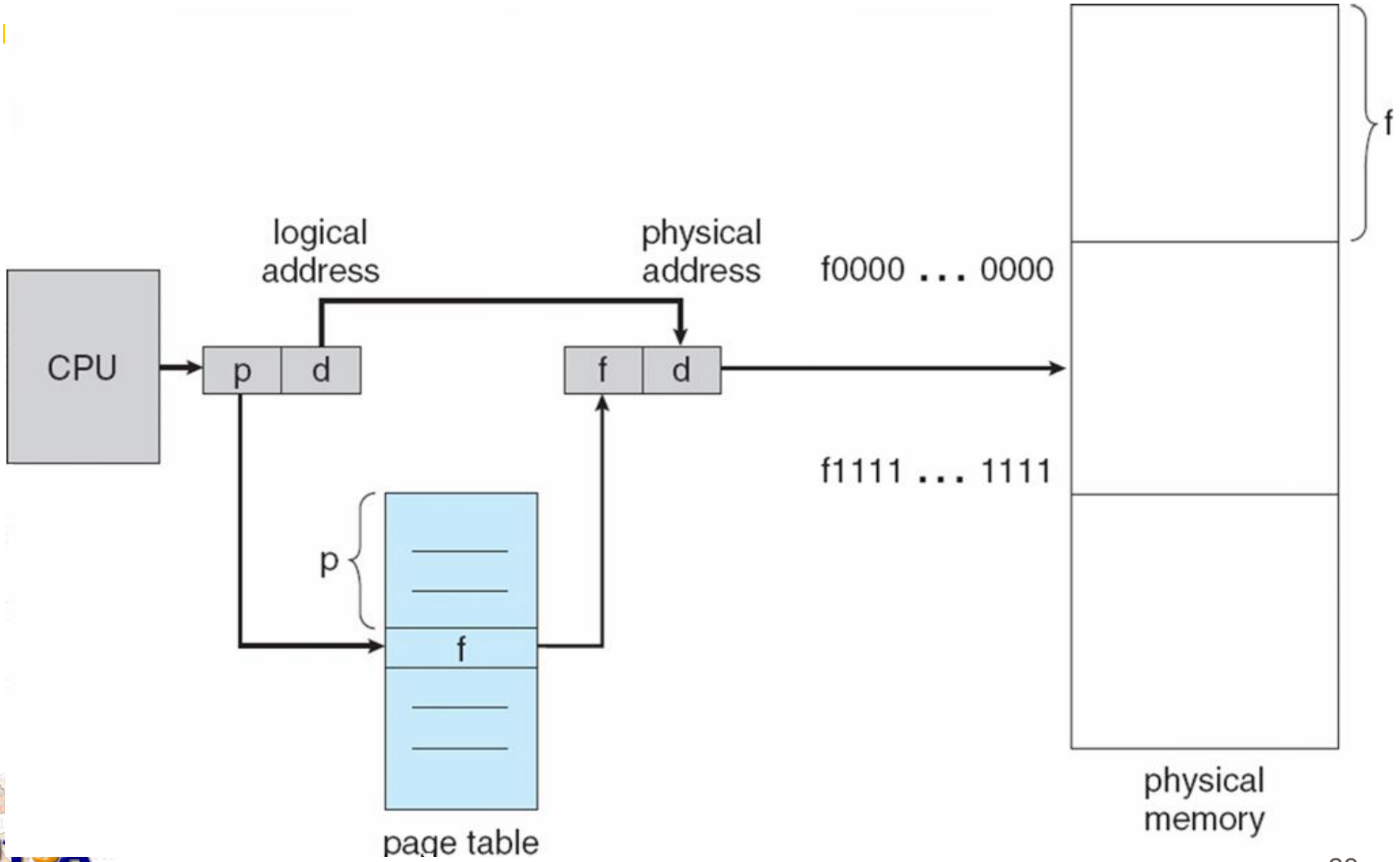
- How many **frames** do we have?
- Suppose the physical address space is 2^b *and frame size* is 2^n
  - Then, the number of frames is $2^b / 2^n$, which is 2^{b-n}

Physical Address (*b* bits) is divided into:

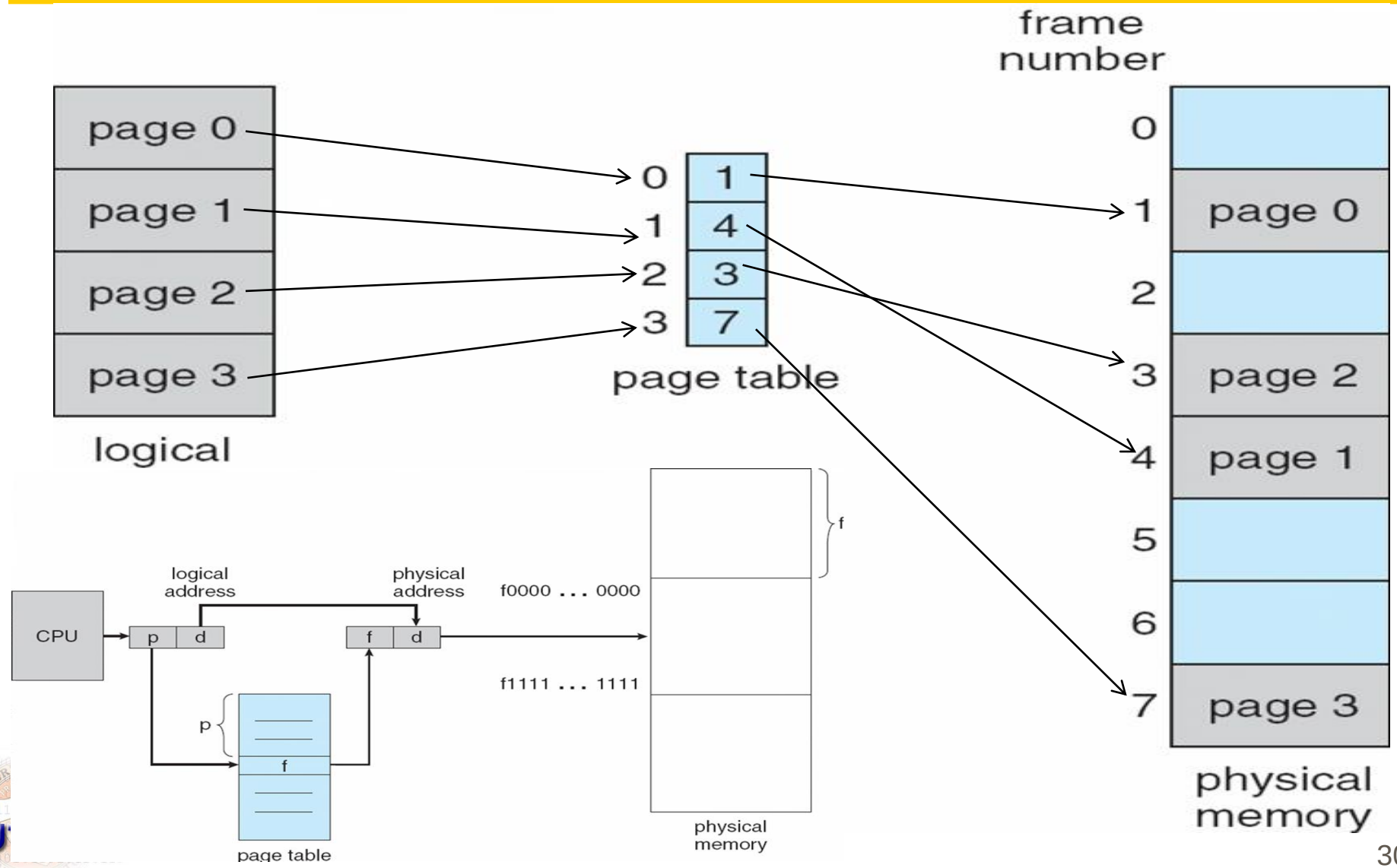
- **Frame number (*f*)** – stored in a *page table* which is indexed by the corresponding page number in logical memory
- **Frame offset (d)** – combined with the frame to define the physical memory address that is sent to the memory unit



Paging Hardware

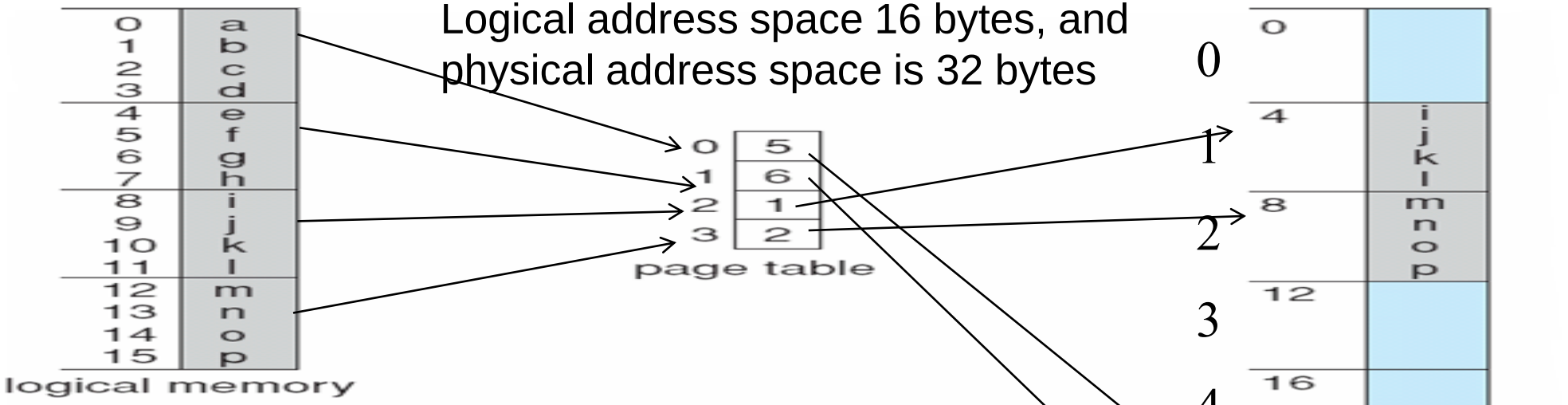


Paging Model of Logical and Physical Memory



Paging Example

Suppose the page size is 4-byte.
Logical address space 16 bytes, and
physical address space is 32 bytes



# of pages = 16/4 = 4

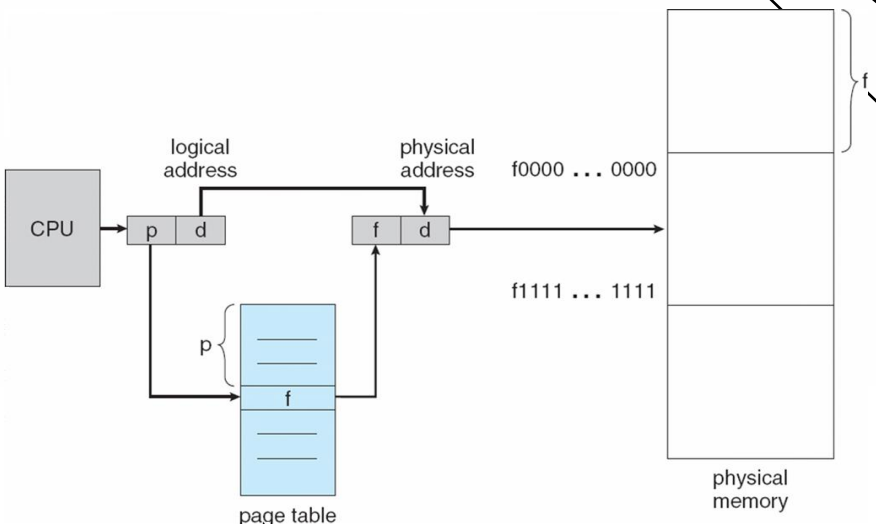
# of frames = 32/4 = 8

Page number (*p*) : ?

Page offset (d) : ?

Frame number (f) : ?

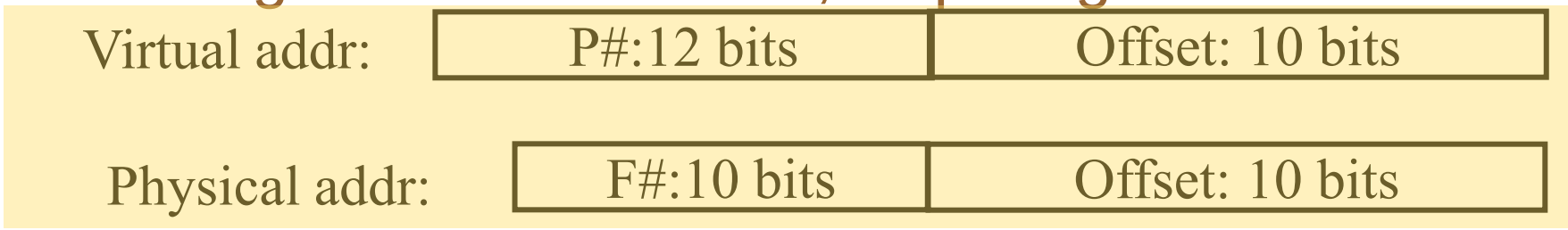
Page table size: ?



Determine Page/Frame Size

- Example:
  - Logical (Virtual) address: 22 bits → 4 MB
  - Physical address (bus width): 20 → 1MB

Option 1: Page/frame size: 1KB, requiring 10 bits



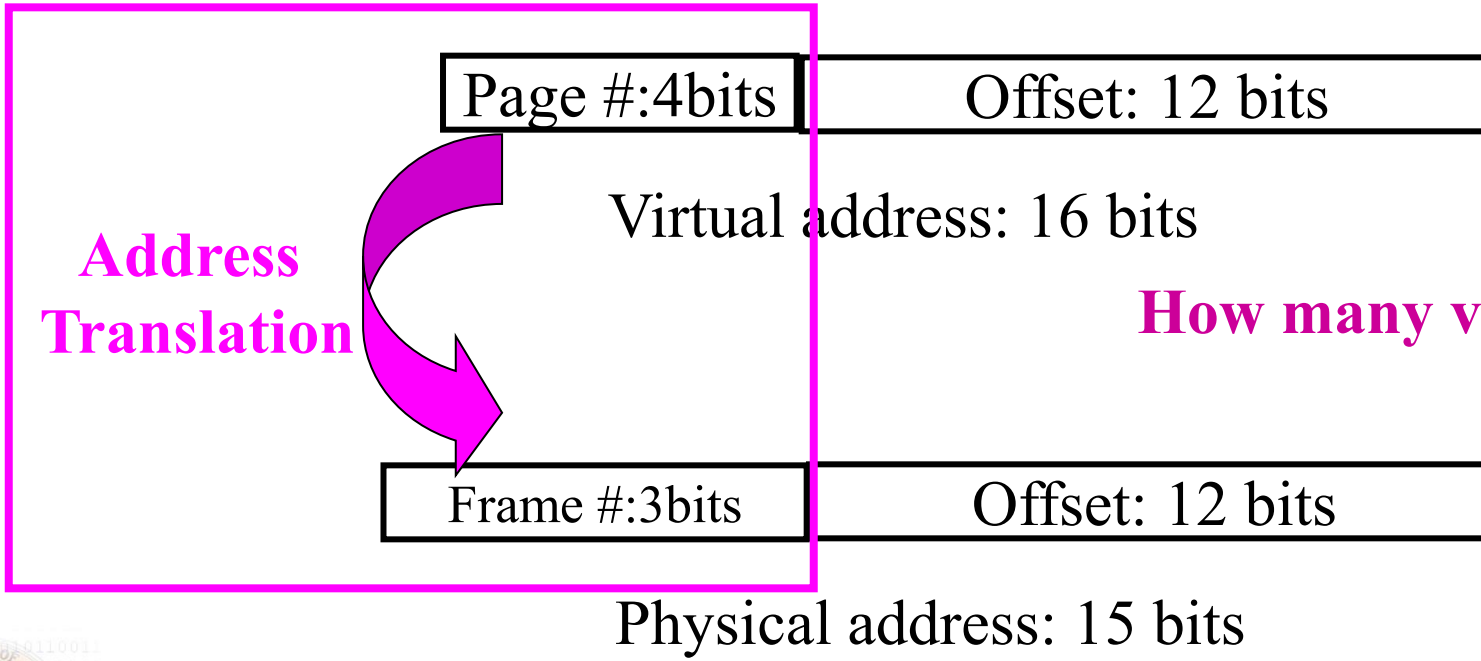
- Number of pages: 12 bits → 4K pages
- Number of frames: 10 bits → 1K frames
- Size of page table:
  - Each page entry must have at least 10 bits
  - 4K * 10 bits = 40 Kbits = 5KB (maximum)

General design principle:
Interplay between page number vs. offset size
Try to fit page table into one frame
Else, balance number of levels

Another Example

- Example:
  - 64 KB virtual memory
  - 32 KB physical memory
  - **4 KB page/frame size → 12 bits as offset (d)**

How about the size of the page table?



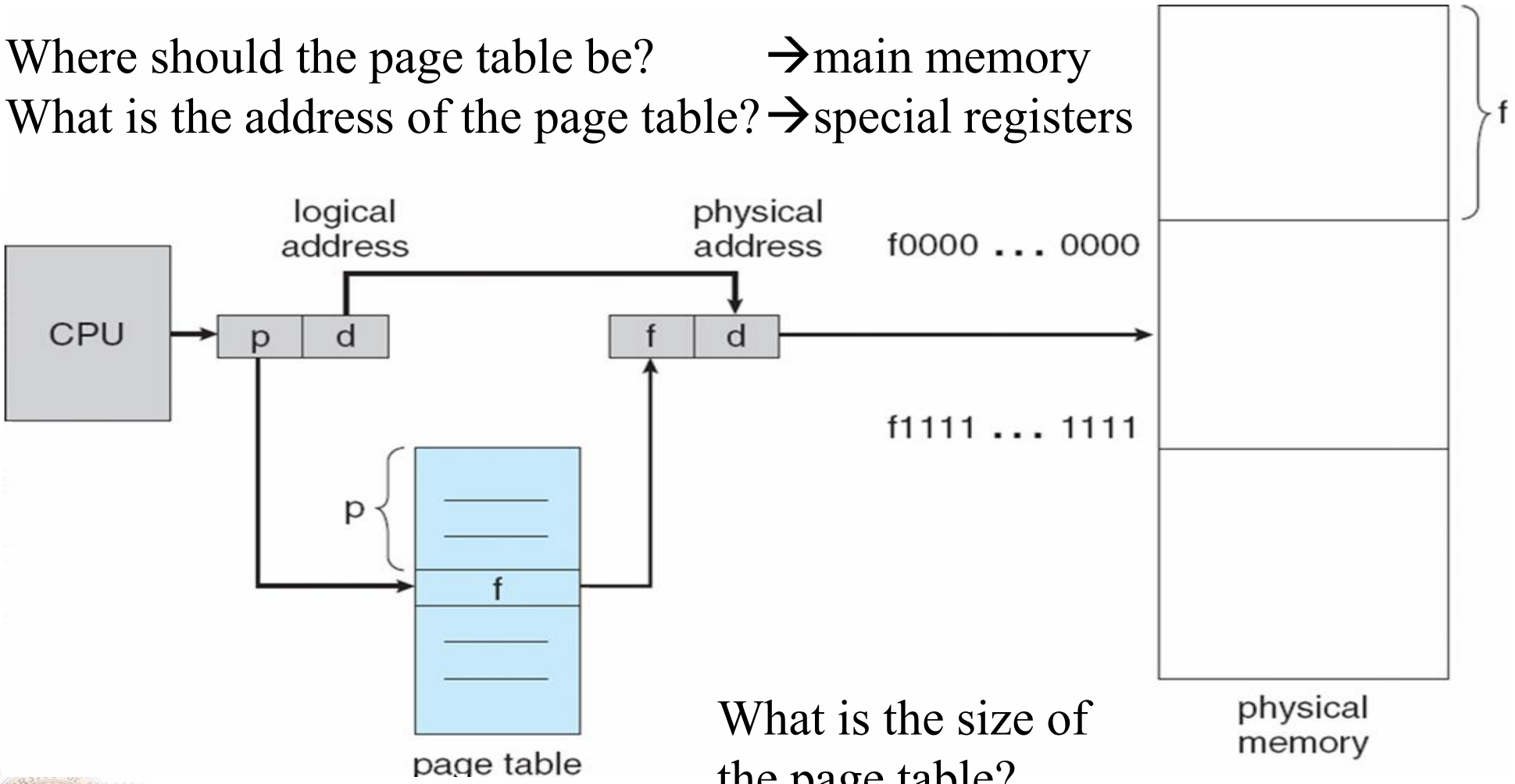
How many virtual pages?

How many physical frames?

Size and Location of Page Table

Where should the page table be? →main memory

What is the address of the page table? →special registers



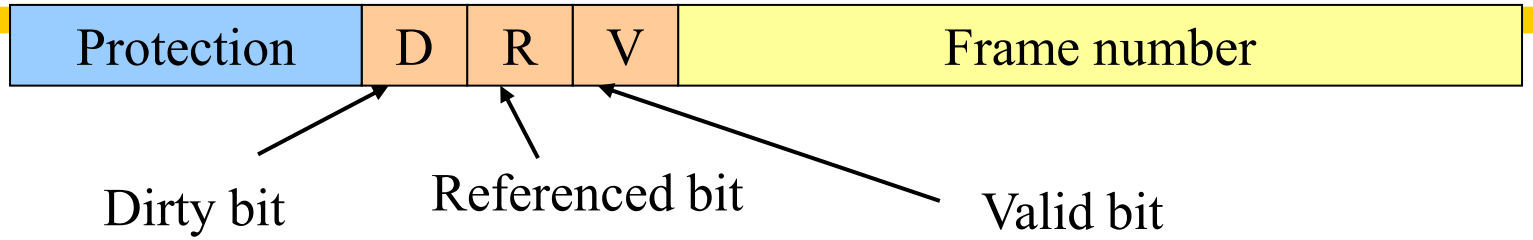
What is the size of the page table?

= 2^p * size of each entry (at least f-bit)

Size of Page/Frame: How Big?

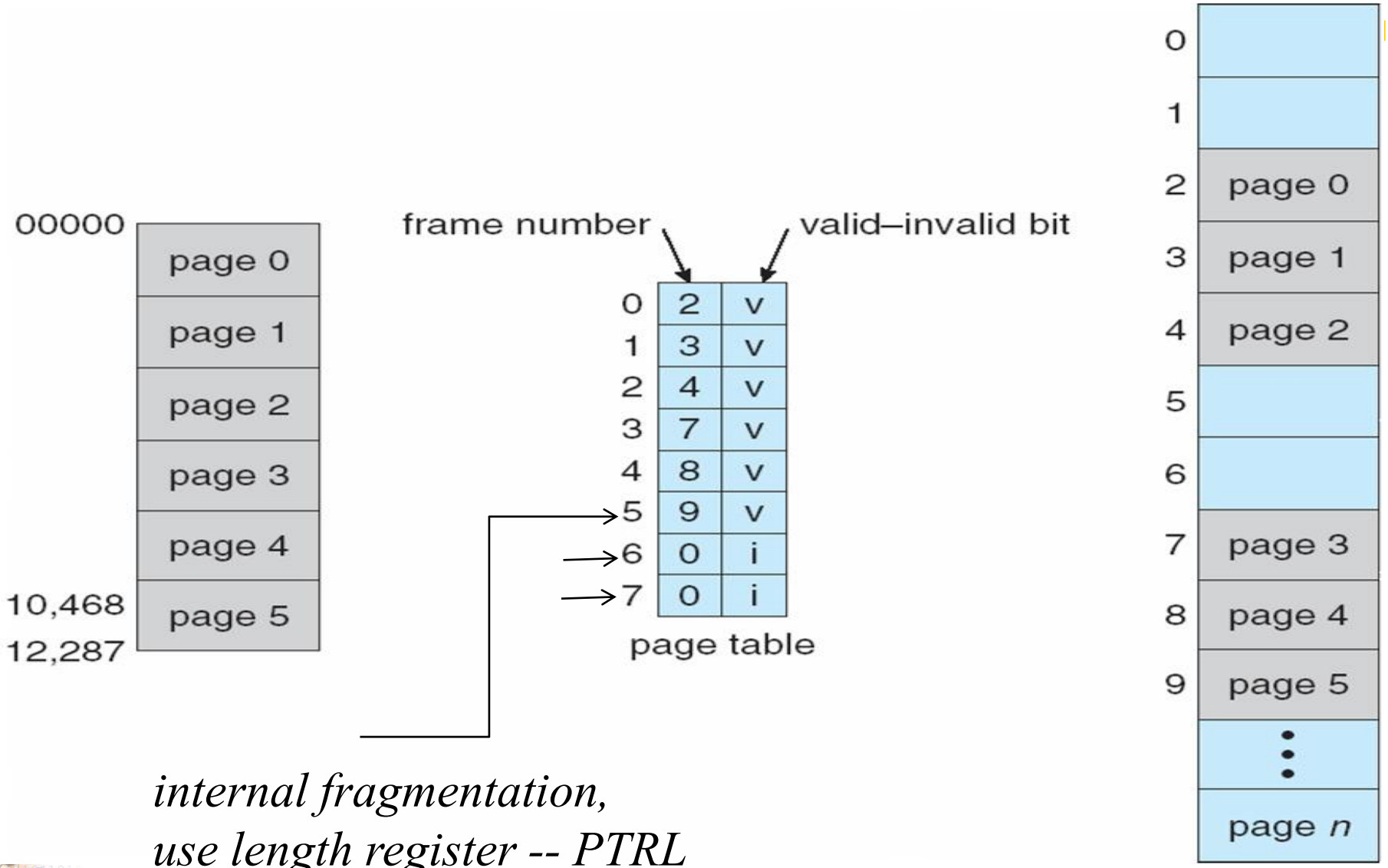
- Determined by **number of bits in offset** (512B→16KB and become larger)
- Smaller pages
  - + Less internal fragmentation
  - + Better fit for various data structures, code sections
  - - Too large page table: spin over more than one frame (need to be continuous due to index), hard to allocate!
- Larger pages
  - + Small page table so less overhead to keep track of them
  - + More efficient to transfer larger pages to/from disk
  - - More internal fragmentation; waste of memory
- **Desing principle: fit page table into one frame**
  - If not, multi-level paging (might be discussed later)

Memory Protection and Other bits



- Several bits might be associated with Page Table Entry (PTE)
  - **Valid-invalid** bit attached for memory protection
    - ✓ "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
  - Referenced bit: set if data on the page has been accessed
  - Dirty (modified) bit: set if data on the page has been modified
  - Protection information: read-only/writable/executable or not
- Size of each PTE is at least frame number plus 2/3 bits
- So, the size of a page table = 2^p * size of PTE

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



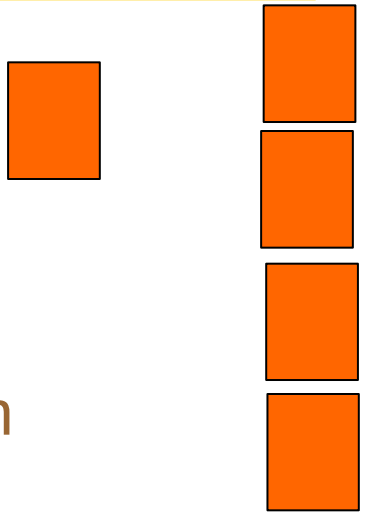
internal fragmentation,
use length register -- PTRL

Paging Pros/Cons of Paging

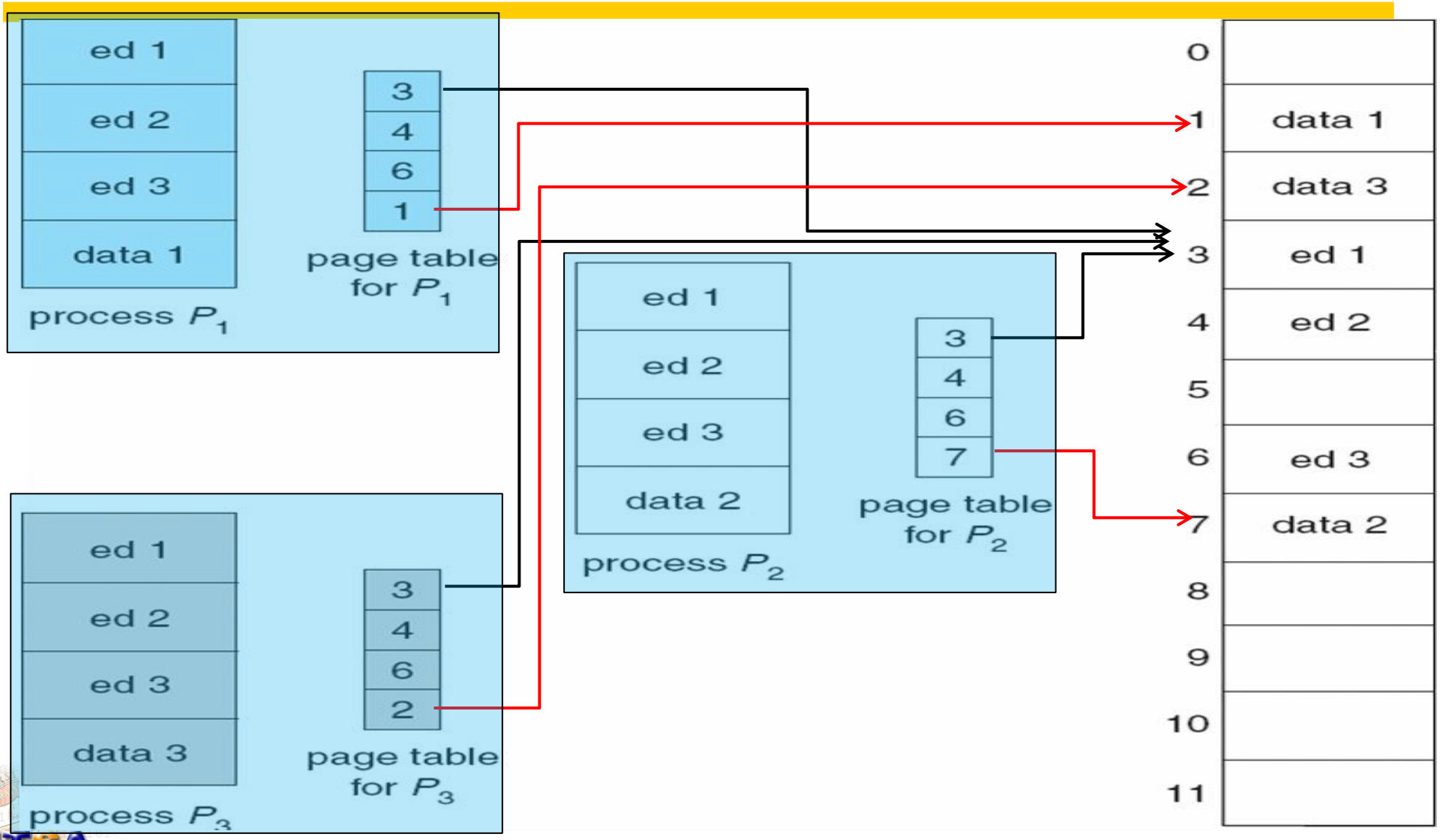
- Pros/Cons
  - + *Allows sharing code among multiple processes*
  - + Provide dynamic relocation
  - + No external fragmentation
  - + Protection (allows access to addresses in the pages in page table)
  - - *Internal fragmentation* (1 page +1 byte require 2 pages)
  - - *Keep track of all free frames*,
  - - Page table for each process (increase context switch time)
- Should we select **big** or **small** pages????
  - Small: *+ less internal fragmentation, - big page table*
  - Big: *+ small page table, + efficient I/O, - more internal fragmentation*
- Needs Hardware support

+ Shared Pages

- **Shared code**
  - One copy of read-only (**reentrant**) code shared among processes (i.e., text editors, compilers, window systems).
  - Shared code must appear in the same location in the logical address space of all processes
- **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



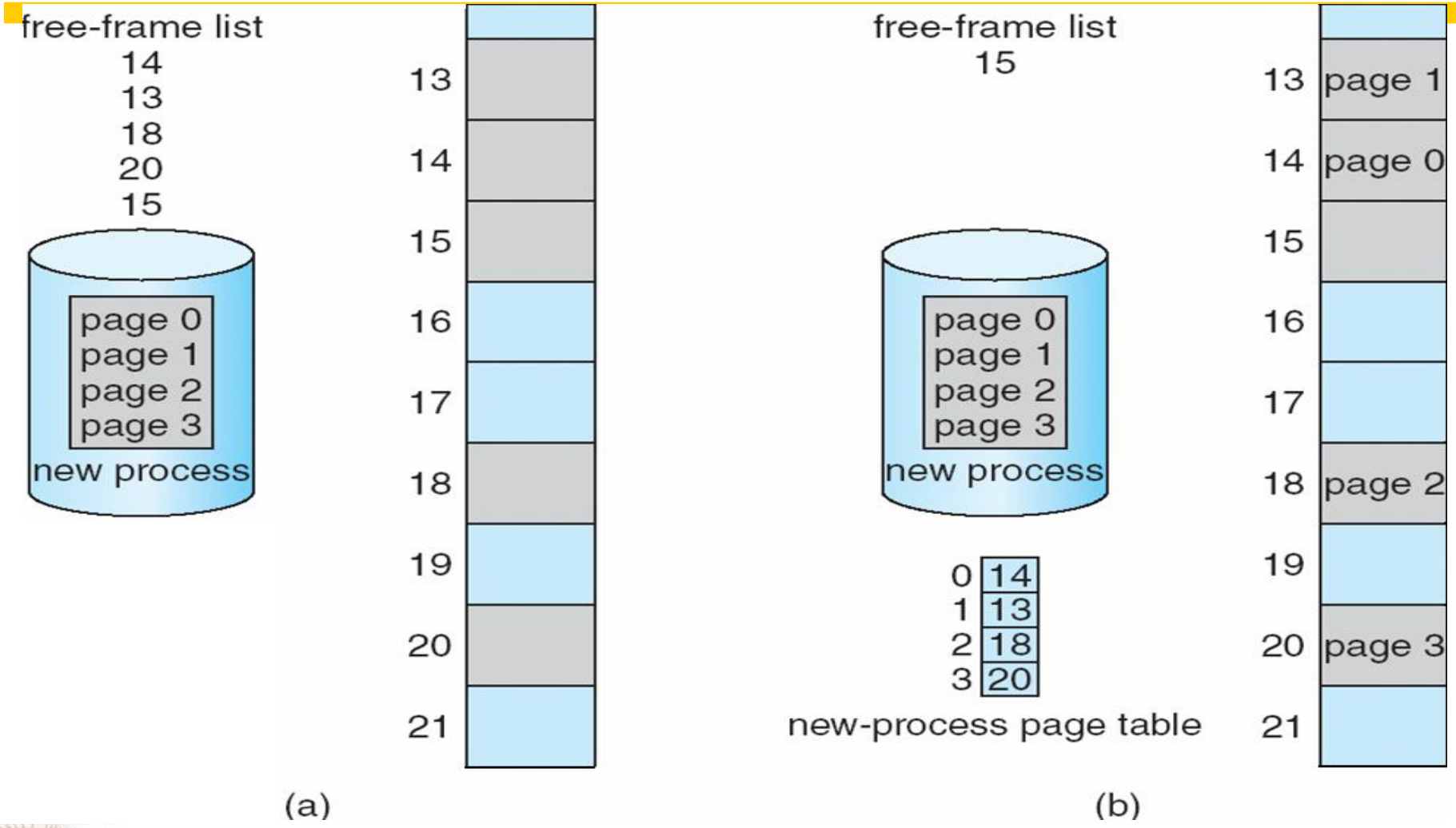
+ Shared Pages Example



- Paging: Internal Fragmentation

- Calculating internal fragmentation
  - Page size = 2K = 2,048 bytes
  - Process size = 72,766 bytes
  - 35 pages + 1,086 bytes
  - Internal fragmentation of 2,048 - 1,086 = 962 bytes
  - Worst case fragmentation = 1 frame – 1 byte
  - On average fragmentation = 1 / 2 frame size
- So, small frame size is desirable!
- But what was wrong with small frame size?

- Free Frames



Before allocation

After allocation

Memory Management

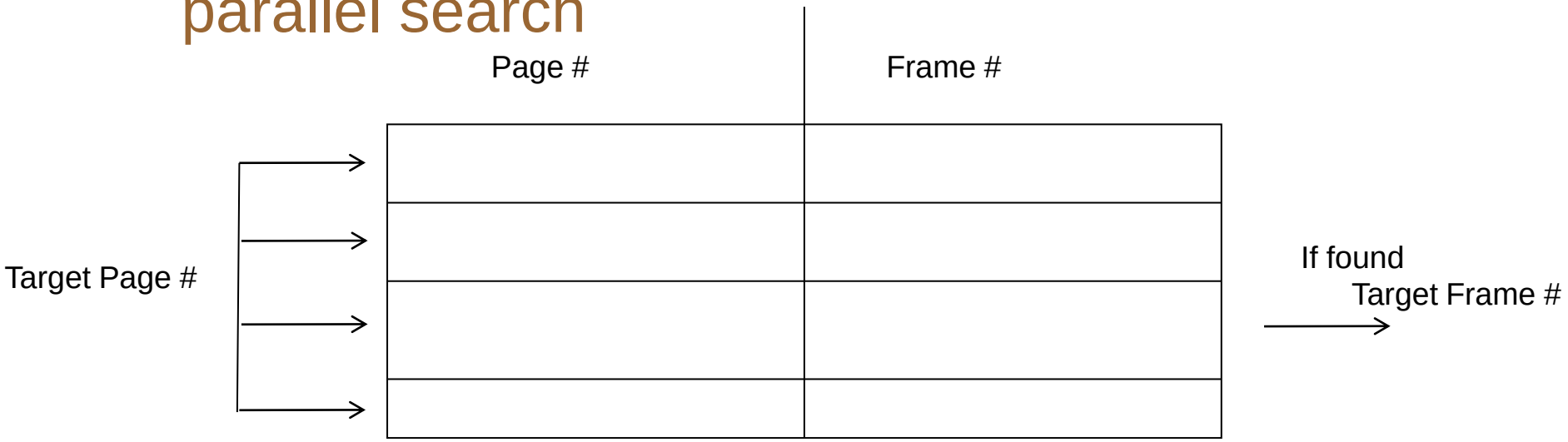
- Background
- Swapping
- Contiguous Memory Allocation and Fragmentation
- Paging
- Page Table implementation, TLB, access time
- Other Page Table Implementations and Structures

Implementation of Page Table

- Where should we store Page table?
  - Registers (fast efficient but limited), main memory, dedicated lookup tables
- Memory
  - **Page-table base register (PTBR)** points to the page table in memory
  - **Page-table length register (PRLR)** 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.
- Dedicated lookup tables
  - 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)** *(see next slide)*
  - Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry – uniquely identifies each process to provide address-space **protection** for that process. If there is no ASID, flush TLB (expensive context SW)

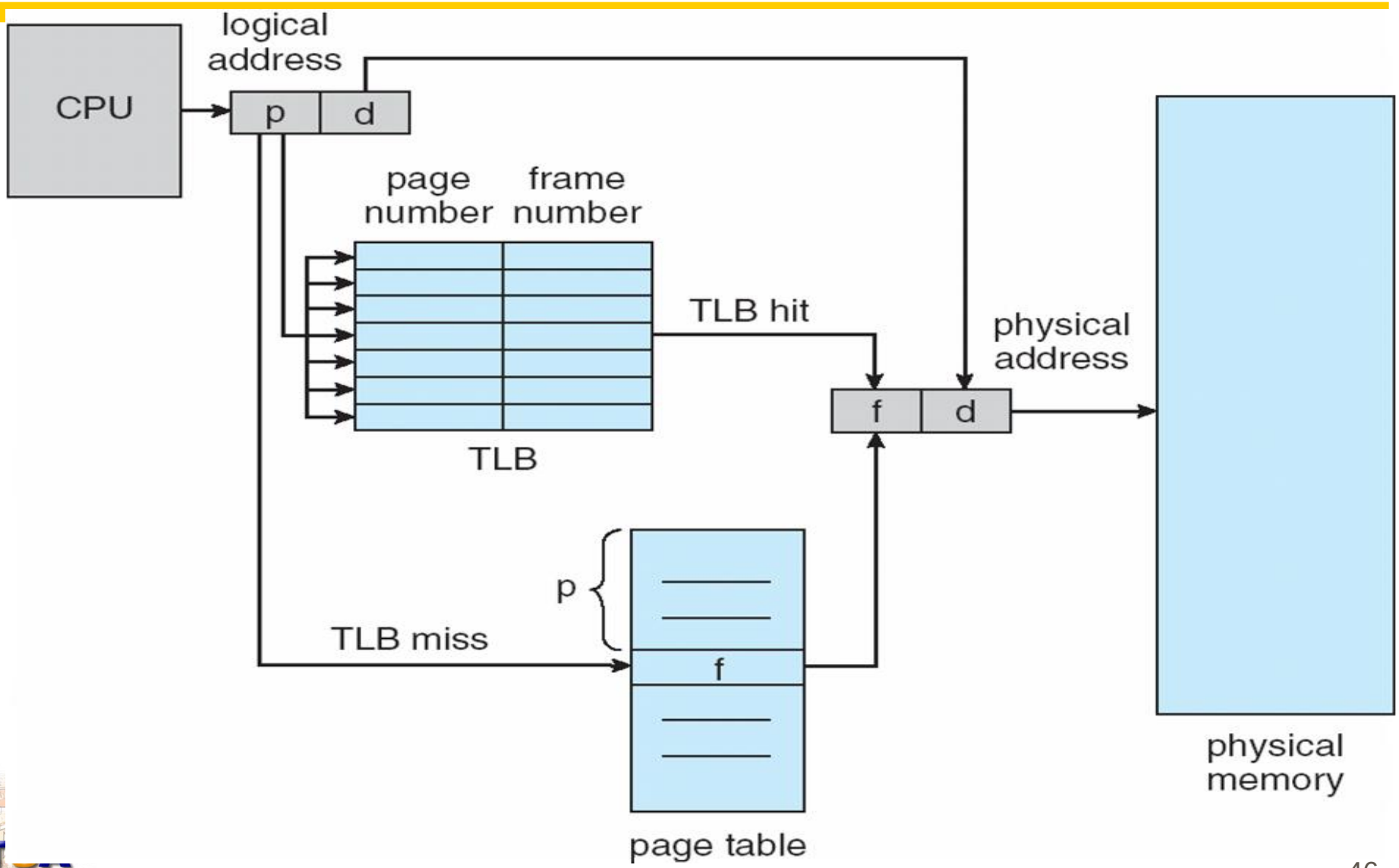
Associative Memory

- Associative memory (**access by content**) – parallel search



- Address translation (p, d)
  - If p is in associative register, get frame # out
  - Otherwise, access the page table in memory, and get frame # and put it into TLB
    - *(if TLB is full, replace an old entry. Wired down entries will not be removed)*

Paging Hardware With TLB



Effective Access Time

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

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

- Associative Lookup = 20 nanosecond
- Memory cycle time is 100 nanosecond microsecond
- Hit ratio = 80%
- **Effective Access Time** (EAT) would be

  $EAT = (100 + 20)\,0.8 + (200 + 20)(1 - 0.8)$

  $= 140$ nanosecond
- 40% slow down
- With 98% hit rate, EAT would be 122 seconds.

Memory Management

- Background
- Swapping
- Contiguous Memory Allocation and Fragmentation
- Paging
- Page Table implementation, TLB, access time
- Other Page Table Implementations and Structures

Page Table size and allocation

- Logical address space varies in the range of 2^{32} to 2^{64}
- Page size varies in the range of (512 B) 2^9 to 2^{24} (16 MB)
- Number of pages = Logical address space ÷ Page size
- Page table should be allocated *contiguously* in memory

Example:

If logical address space is 2^{32} = 4GB and page size is 2^{12} = 4KB

then number of pages will be $2^{32} / 2^{12} = 2^{20} = 1M \approx 10^6$

if each entry consist of 4 Bytes then

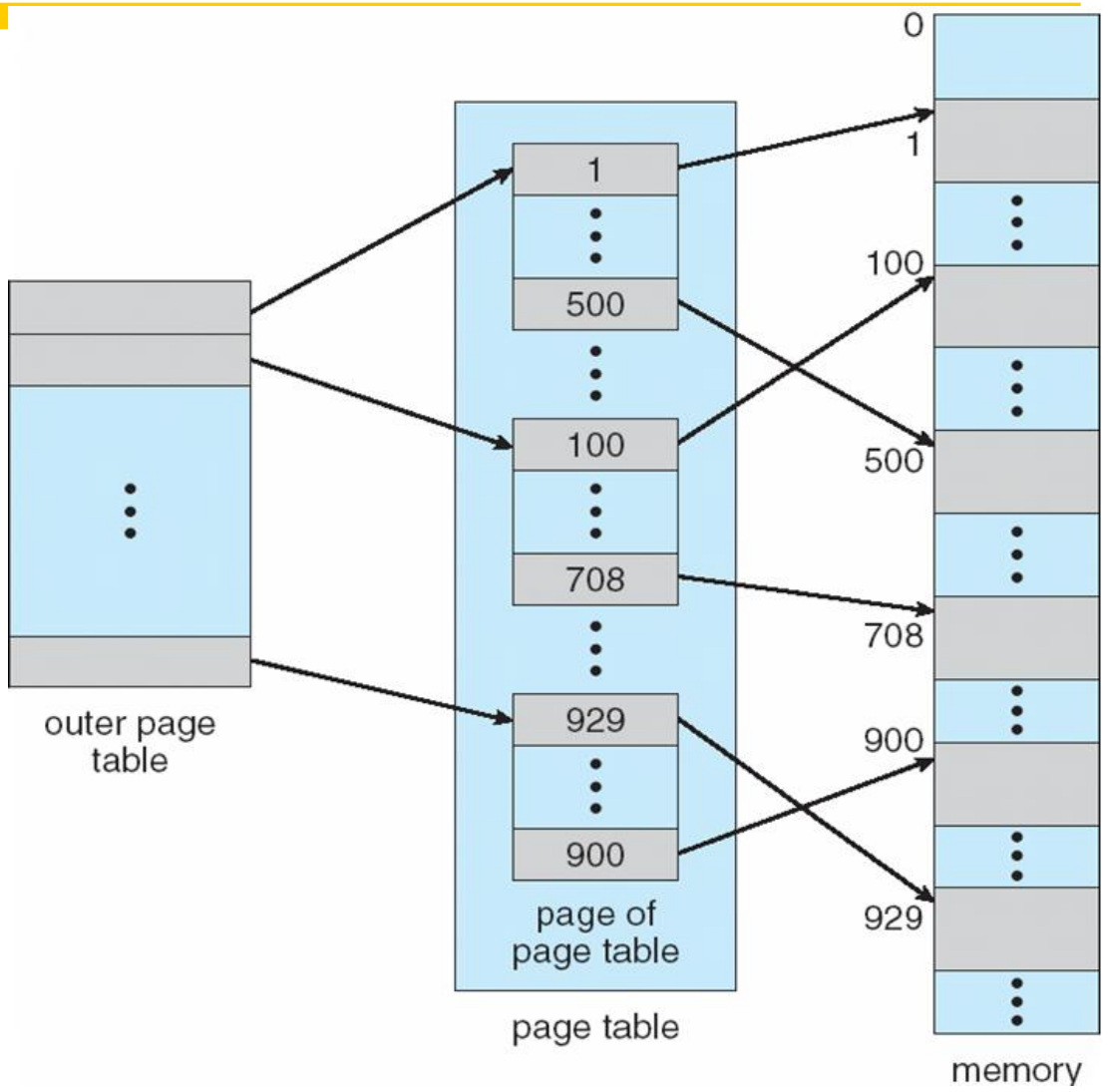
the page table size will be 4MB for each process,

So, we need 4MB/4KB = 2^{10} = 1K frames for page table

But we may not be able to allocate that many frames contiguously!

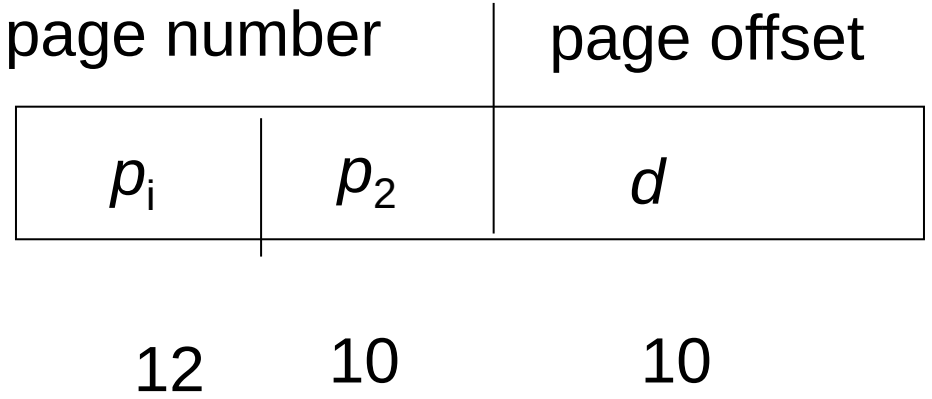
Hierarchical Page Tables

- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
- Why is it good/bad?
  - + We don't have to allocate all levels initially
  - + They don't have to be continuous
  - - how about memory access



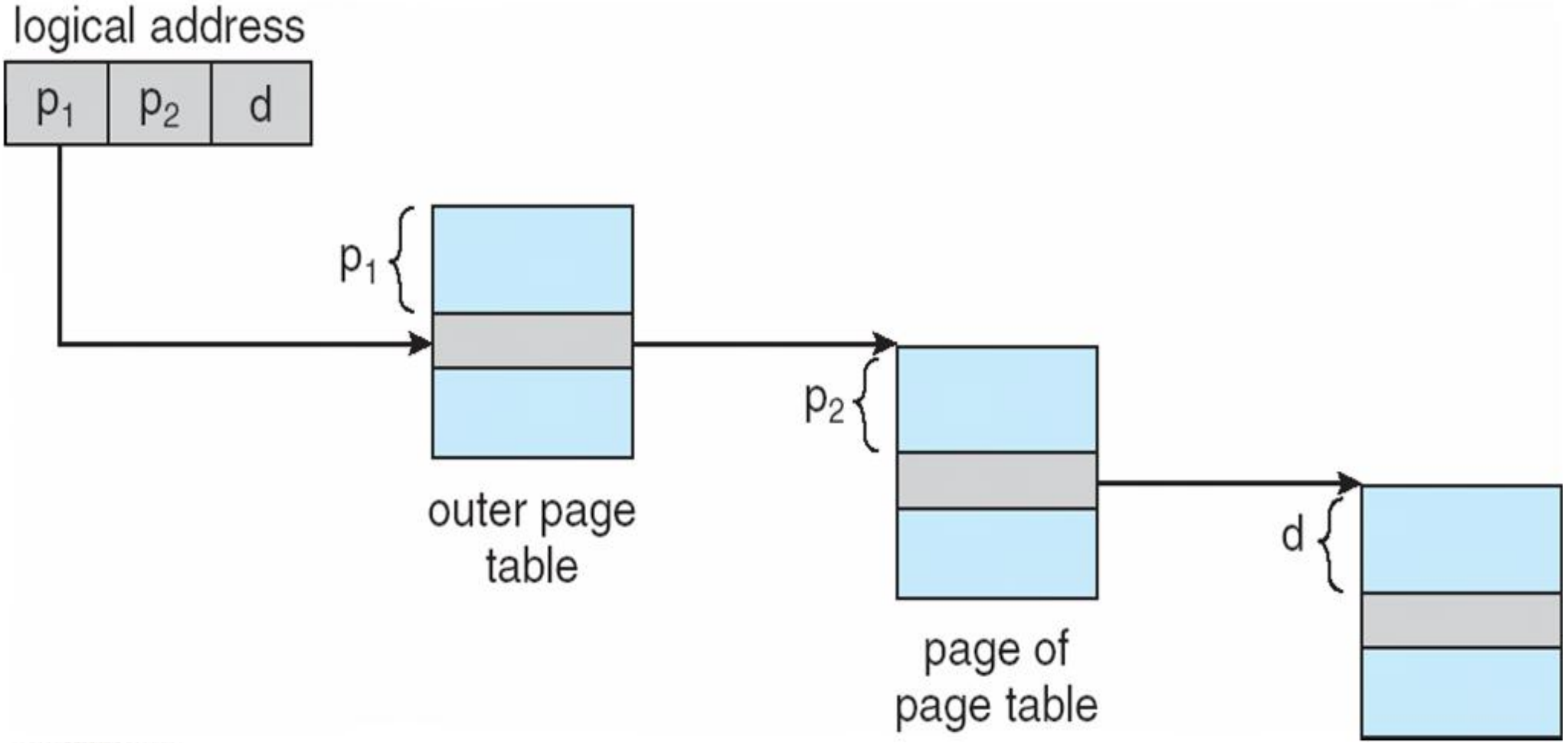
Two-Level Paging Example

- 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:

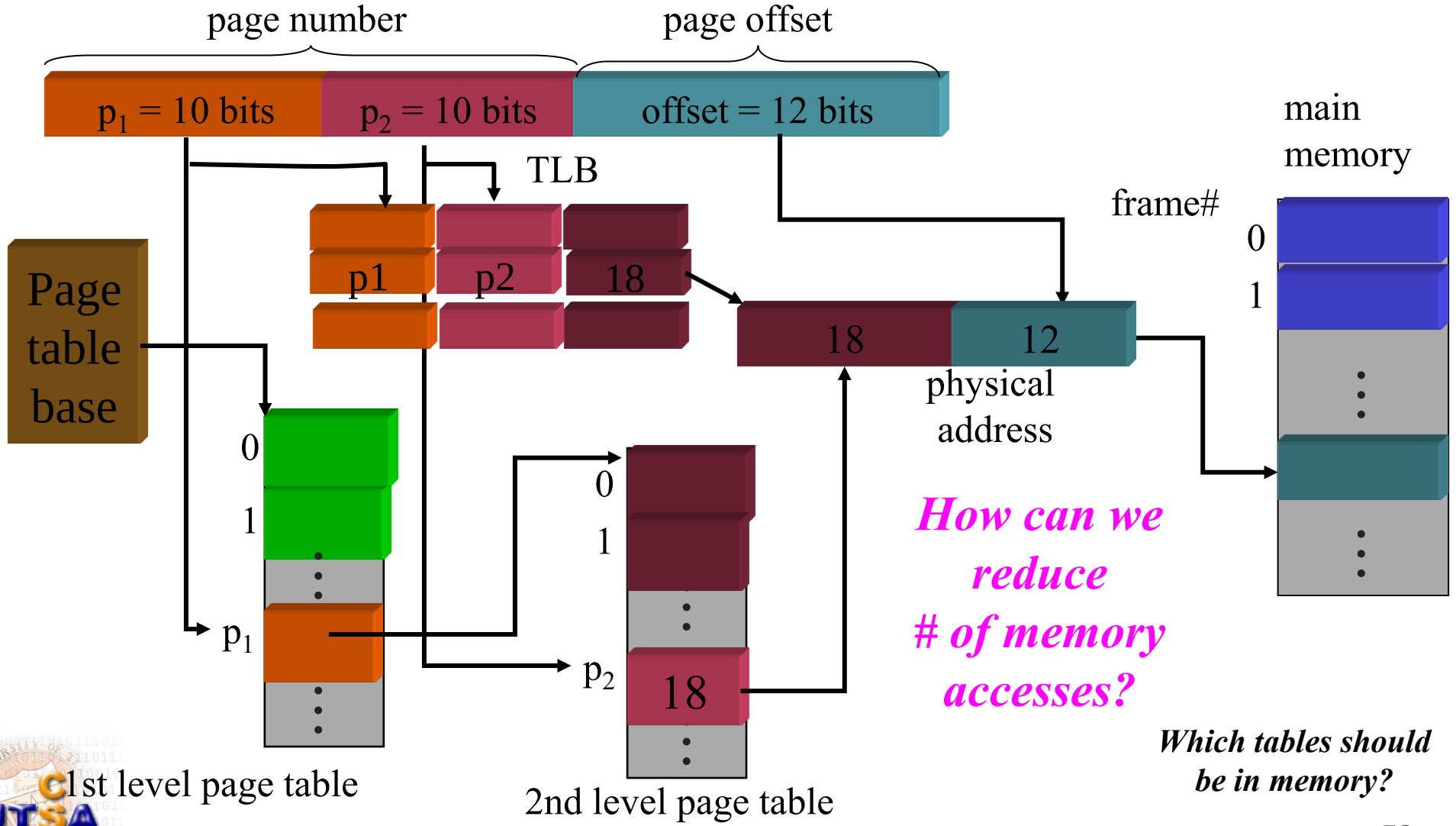


where p_i is an index into the outer page table, and p_2 is the displacement within the page of the outer page table

Two-Level Address-Translation Scheme



Two-level Address Translation with TLB



Memory Requirement of Page Tables

- Only the **1st level page table** and ***the required*** 2nd level page tables need to be in memory

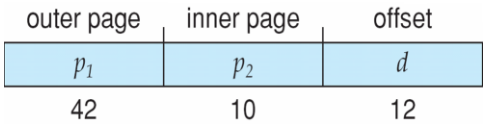
For example:

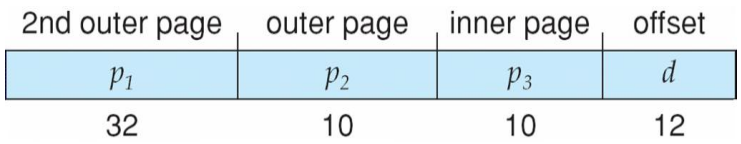
Suppose we have 1GB memory and 32-bit logical (virtual) address and 4KB page size (12-bit). Now we want to run a process with size of 32 MB, how many pages do we need?

- 32MB/4KB → 8K virtual pages (***minimum***) to load this process
- Assuming each table entry is 4 Bytes, then one page can store 1K page table entries. So we need at least 8K/1K = **8** pages, which will be used as second level pages….
- We need a first level page table, for which we will allocate **one** page
- So overall, we need 9 pages (9 X4KB = **36 KB**) to maintain the page tables

How Many Levels are Needed?

- New architectures: 64-bits address?
  - Suppose 4KB page(frame) size (12-bit offset)
  - Then we need a page table with $2^{64} / 2^{12} = 2^{52}$ entries!!!!
  - If we use two-level paging, then inner page table could be one page, containing 1K entries (10-bit) *(assuming PTE size is 4 bytes)*
  - So the outer page needs to contain 2^{42} entries !!!
  - Similarly, we can page the outer page, giving us three-level paging
  - If we continue this way, we will have 6-level paging *(7 memory accesses)*



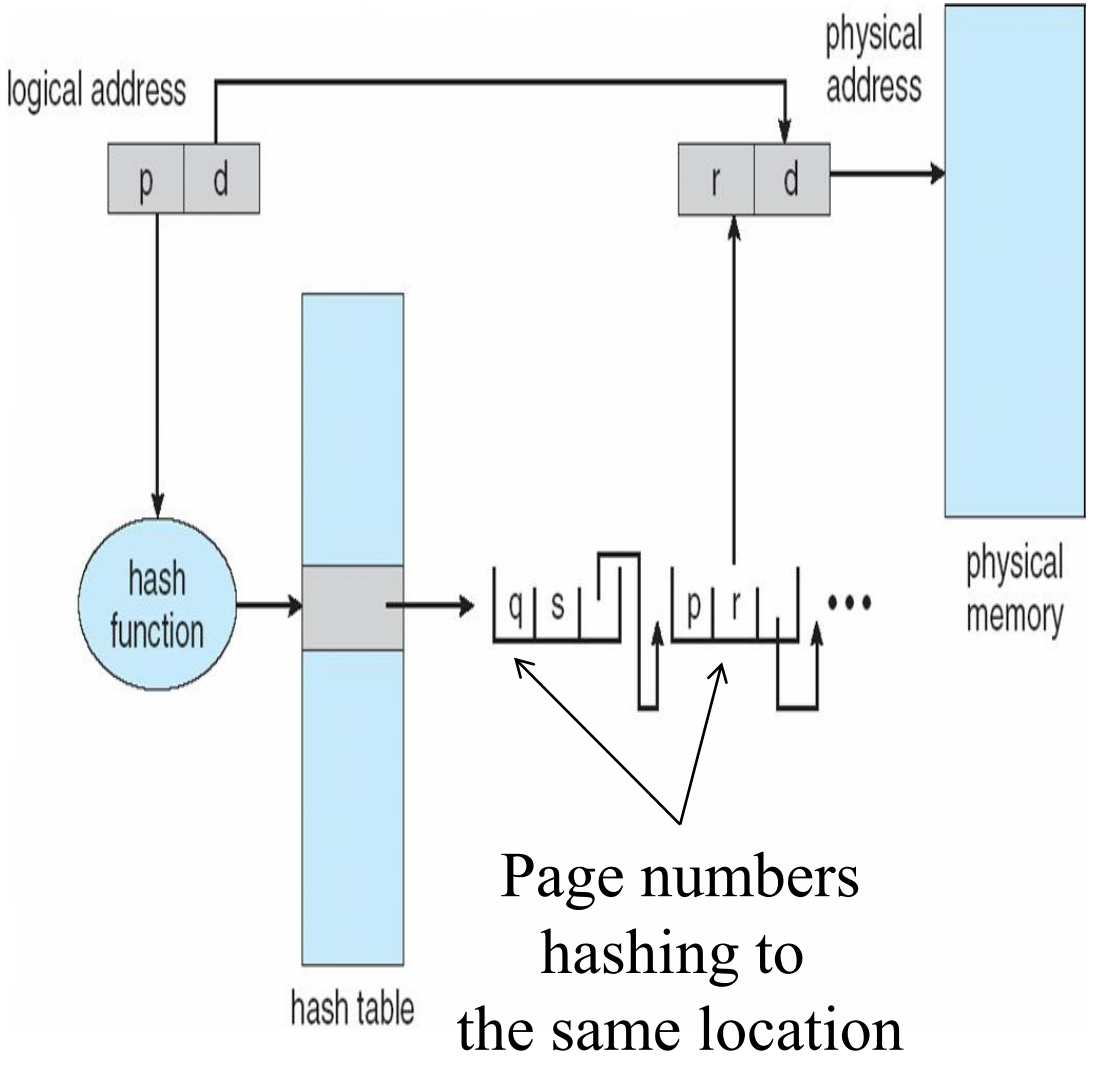


- Problems with multiple-level page table
  - One additional memory access for each level added ***(if not in TLB)***
  - Multiple levels are possible; but prohibitive after a few levels

Is there any other alternative to manage page/frame?

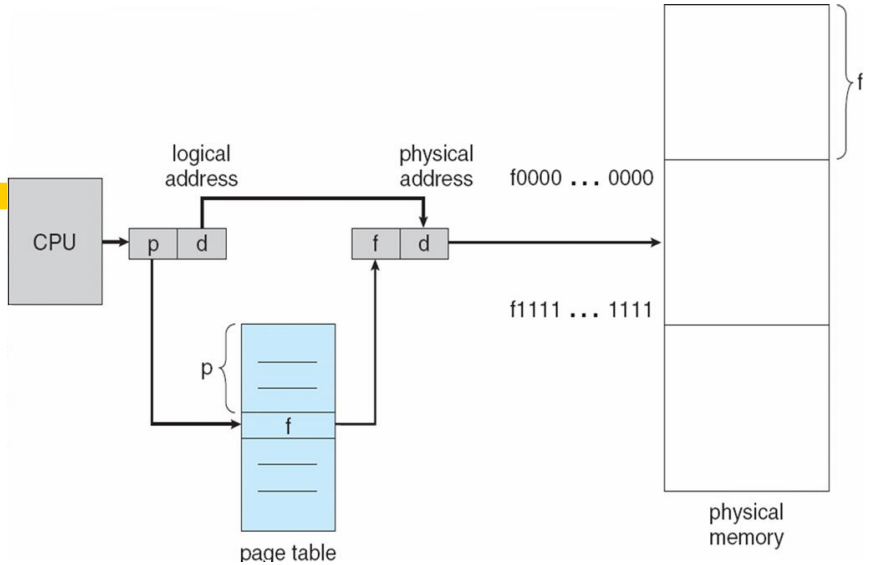
Hashed Page Tables

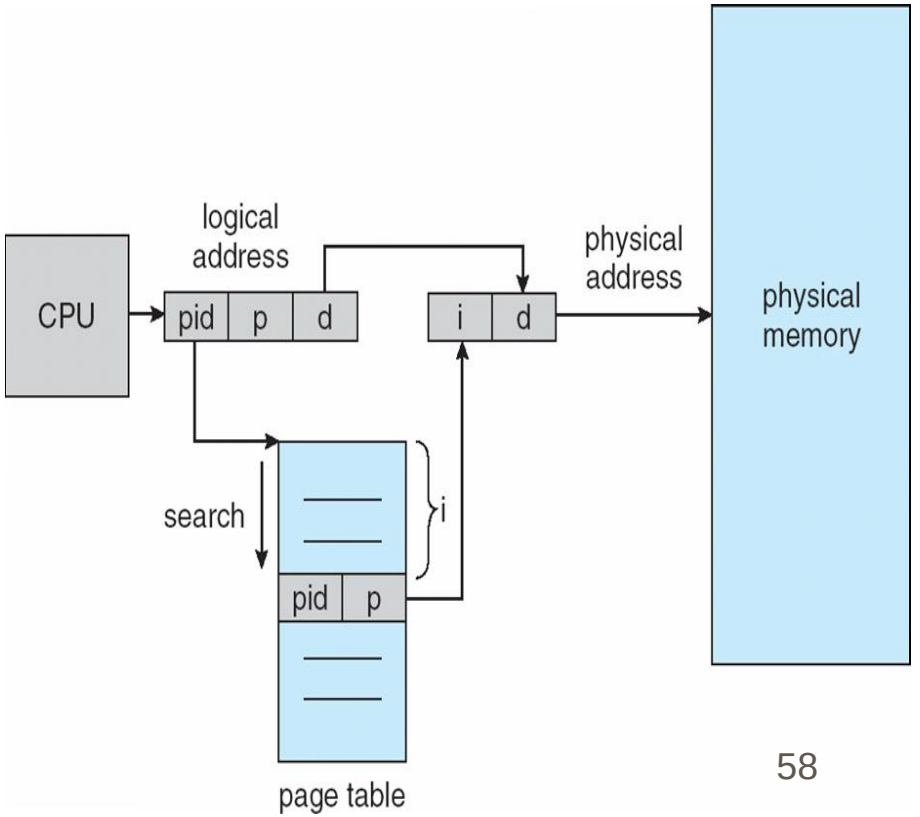
- The virtual page number is hashed into a page table that contains a chain of elements hashing to the same location
- Virtual page numbers are compared in this chain searching for a match. If a match is found, the corresponding physical frame is extracted
- Clustered page tables, each entry refers to several pages…



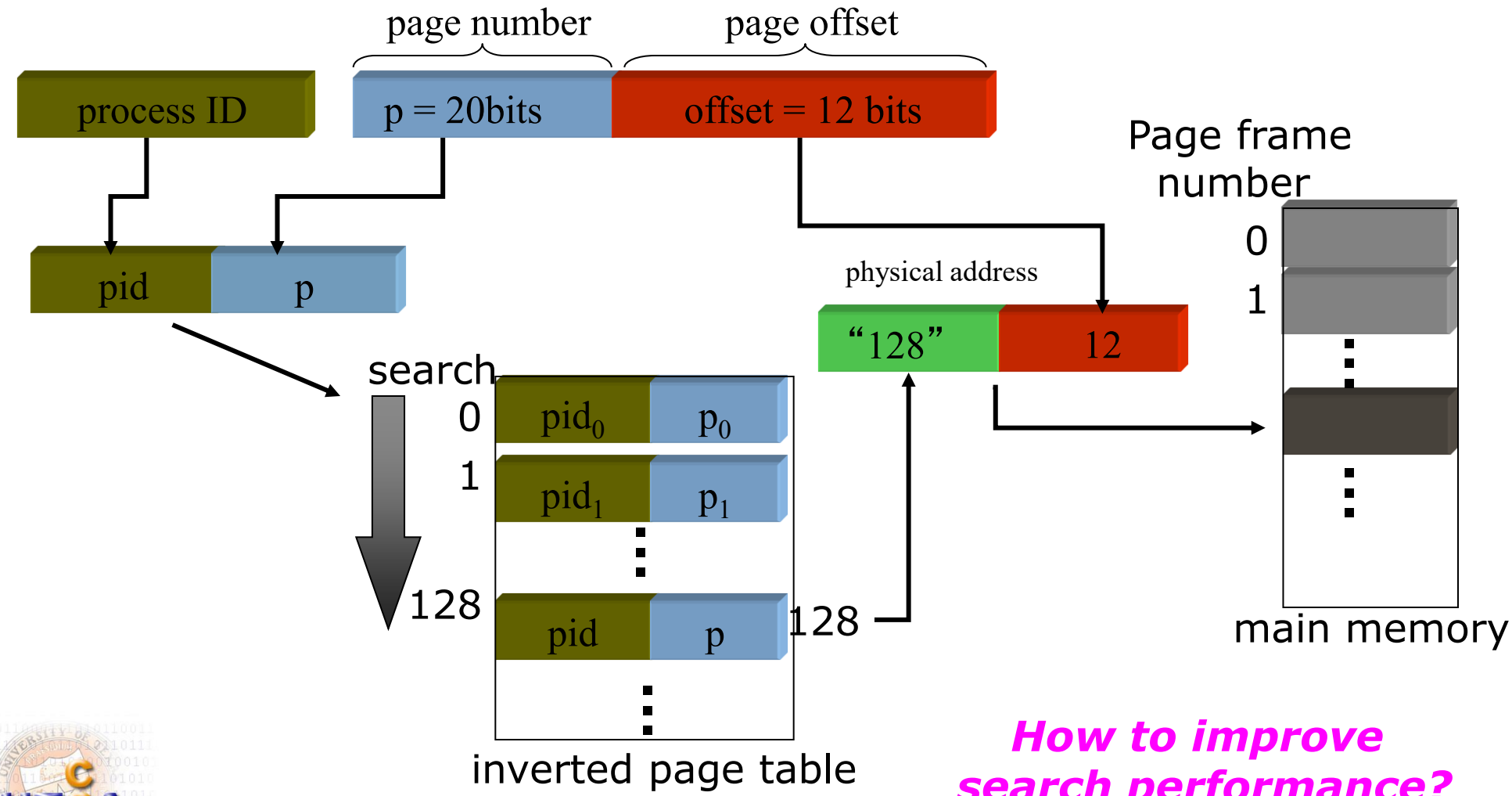
Inverted Page Table

- One entry for each real page (frame) 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. Also hard to implement shared memory
- Use hash table to limit the search to one — or at most a few — page-table entries





Inverted Page Table (cont.)



How to improve search performance?

Hashing for Inverted Page Table

- Hashing function
  - Take virtual page number and process ID
  - Hash value indicates the index for ***possible*** PTE
  - Compare the PTE with virtual page # and PID
  - If the same: hash value is the frame number
  - If not ?

- Confliction of hash function
  - Next PTE until either the entry is found or a limit is reached
  - Second hash function/value

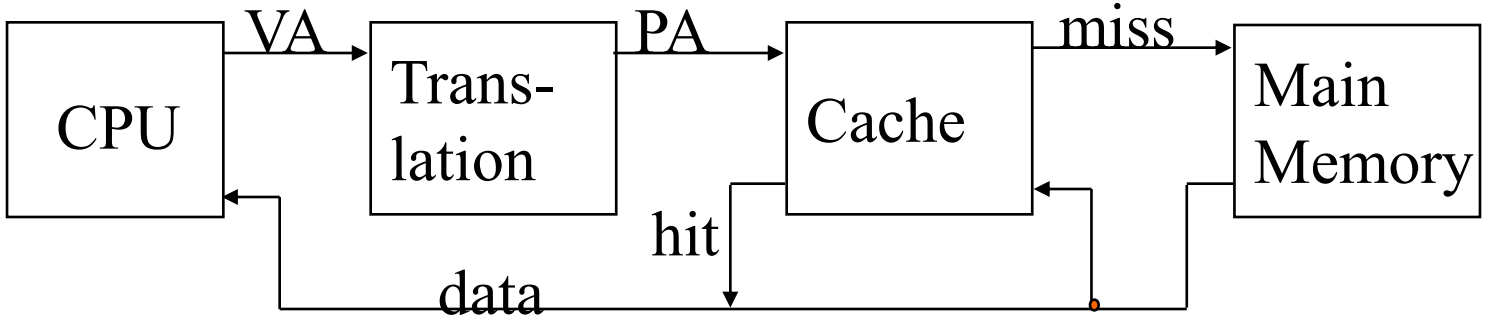
Summary

- Memory Management Background
  - Logical vs. physical addresses
  - Address binding
- Partitions: base + limit registers
- Swapping
- Contiguous Memory Allocation and Fragmentation
- Paging
- Translation lookaside buffer (TLB)
  - Effective memory access time
- Page Table Structures
- Segmentation, and Pentium and Linux page tables

Skip the rest

EXTRAS

Integrating VM and Cache



- Most caches "physically addressed"
  - Accessed by physical addresses
  - Allows multiple processes to have blocks in cache at same time else context switch == cache flush
  - Allows multiple processes to share pages
  - Cache doesn't need to be concerned with protection issues
    - ✓ Access rights checked as part of address translation
- Perform address translation before cache lookup
  - Could involve memory access itself (to get PTE)
  - So page table entries can also be cached

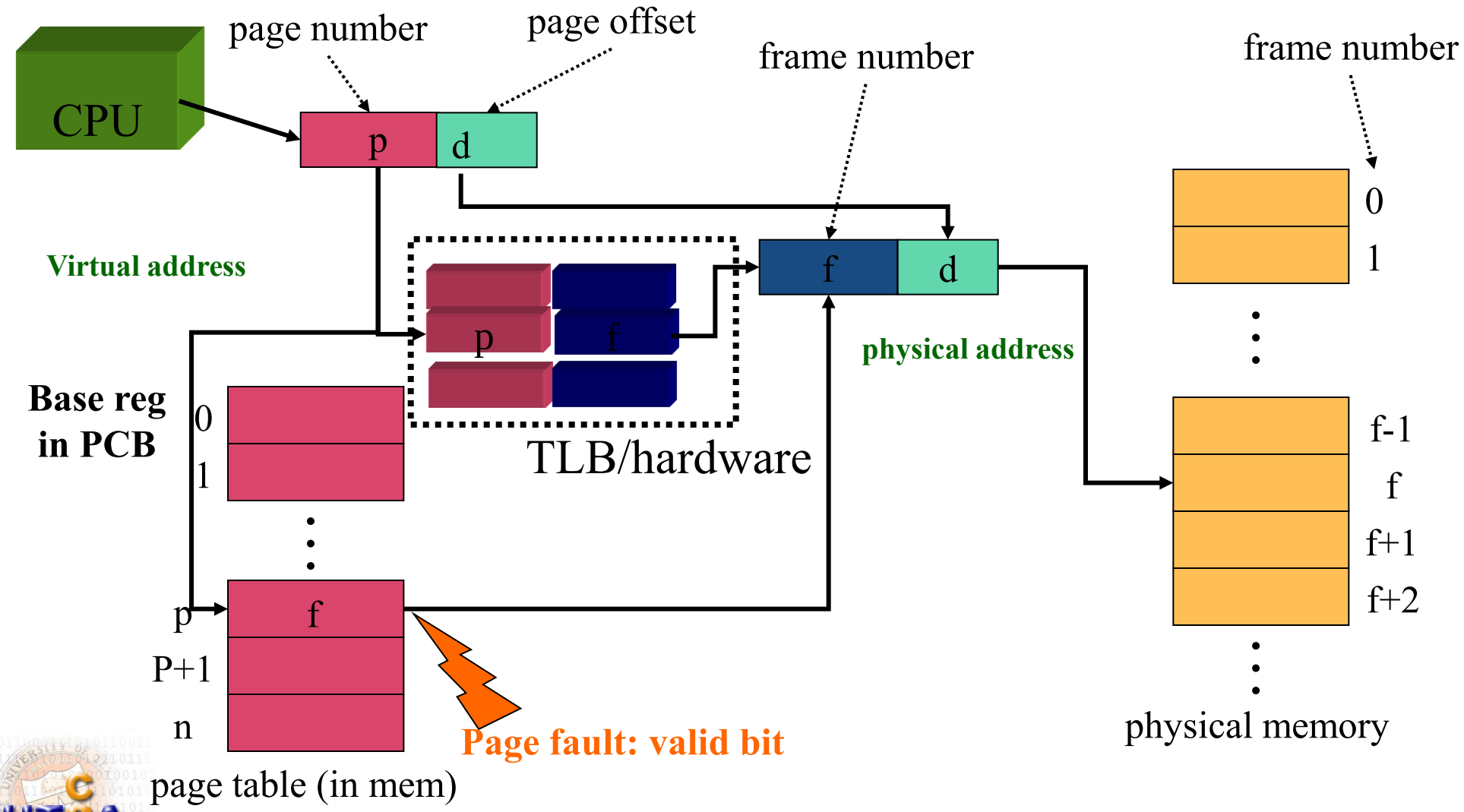
Translation Lookaside Buffer (TLB)

- **Small Hardware**: fast
- Store recent accessed mapping of page → frame (64 ~ 1024 entries)
- If desired logical page number is found, get frame number from TLB
- If not,
  - Get frame number from page table in memory
  - Use standard cache techniques
  - Replace an entry in the TLB with the logical & physical page numbers from this reference
  - Contains complete page table entries for small number of pages

Logical page #	Physical frame #
8	3
unused	
2	1
3	0
12	12
29	6
22	11
7	4

Example TLB

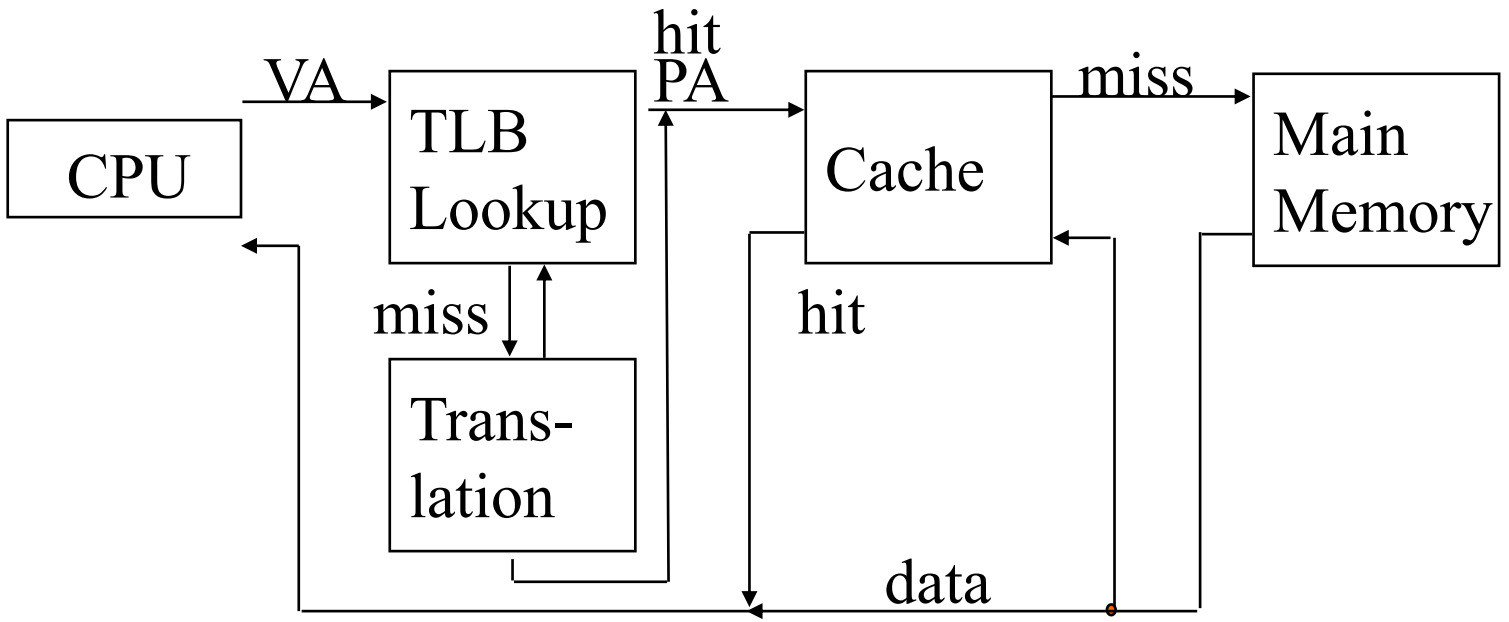
Address Translation with TLB



What happens when cpu performs a context switch?

Integrating TLB and Cache

- "Translation Lookaside Buffer" (TLB)



Understanding the workflow!!!

The procedure of memory access

① Uses special associative cache (TLB) to speed up translation

② Turns to full page translation tables in memory, if TLB misses. This is much slower.

③ Falls back to OS if page translation table misses

- Such a reference to an unmapped address causes a page fault

④ If the memory address is fetched, check whether the block in cache or not.

⑤ If yes, getting the data from the cache.

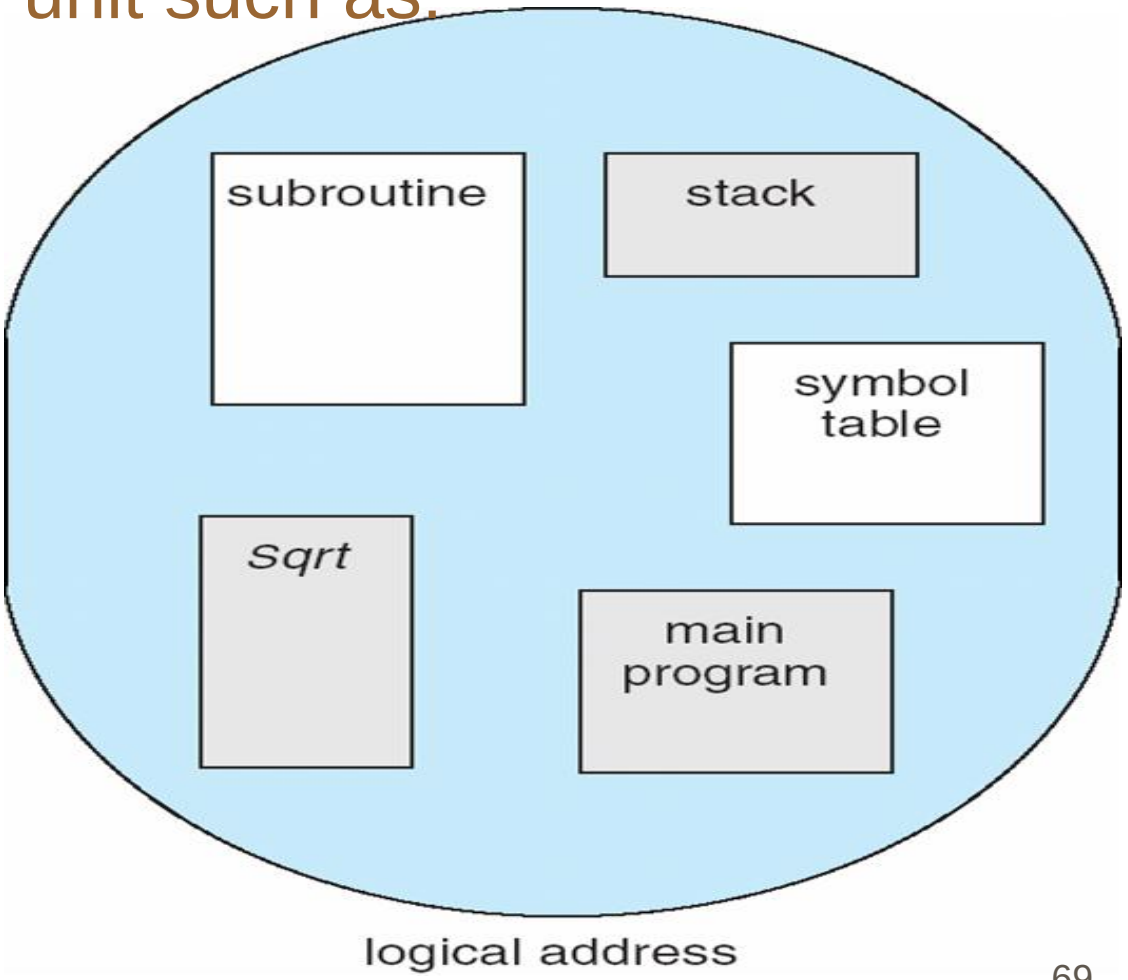
⑥ Otherwise, fetch the cache line from the memory into the cache.

Memory-management scheme that supports user view of memory
Collection of variable size segments

SEGMENTATION

Segmentation

- A user prefers to see a program as a collection of segments
- A segment is a logical unit such as:
  - main program
  - procedure
  - Function
  - Method
  - Object
  - local variables,
  - global variables
  - Common block
  - Stack
  - symbol table
  - arrays

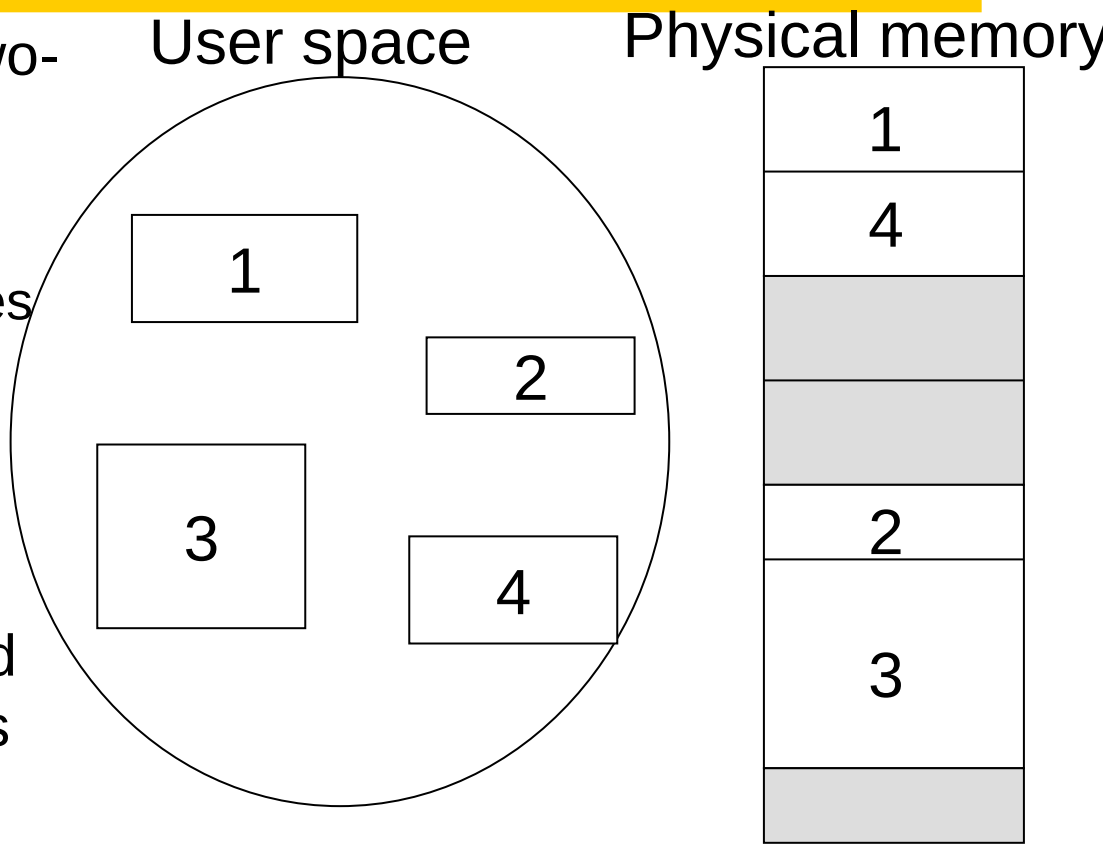


Logical View of Segmentation

- Logical address consists of a two-tuple:

  <segment-number, offset>

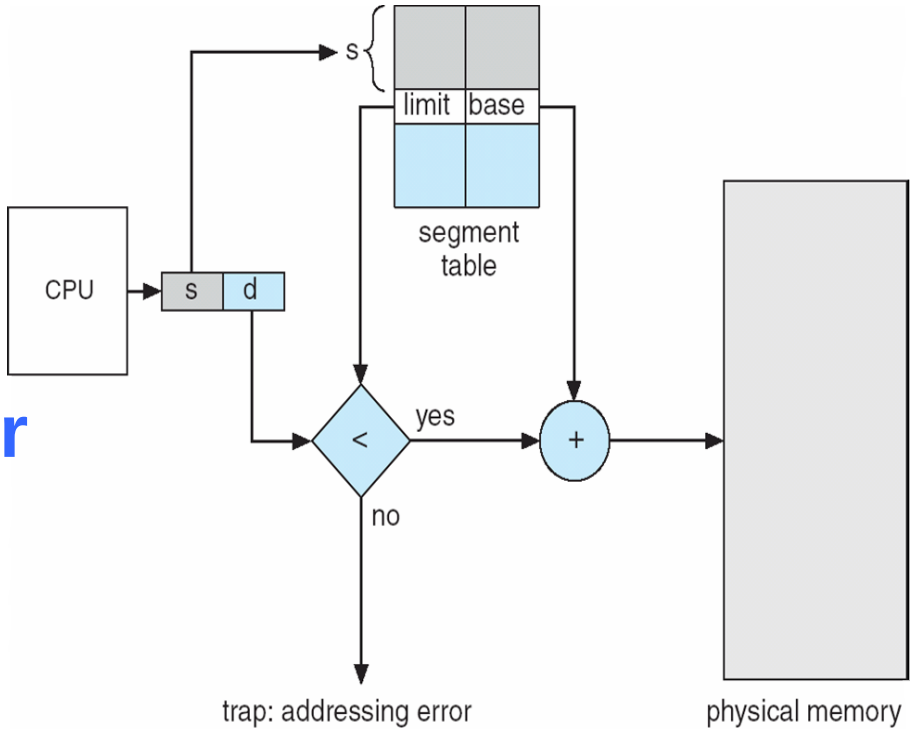
  - For example, C compiler creates different segments for code, global variables, heap, stack, standard lib
- These two-dimensional addresses need to be translated (mapped) to physical addresses



- Looks like paging, but
  - In paging user specifies a single address, which is partitioned by the hardware
  - In segmentation user specifies a segment name and an offset

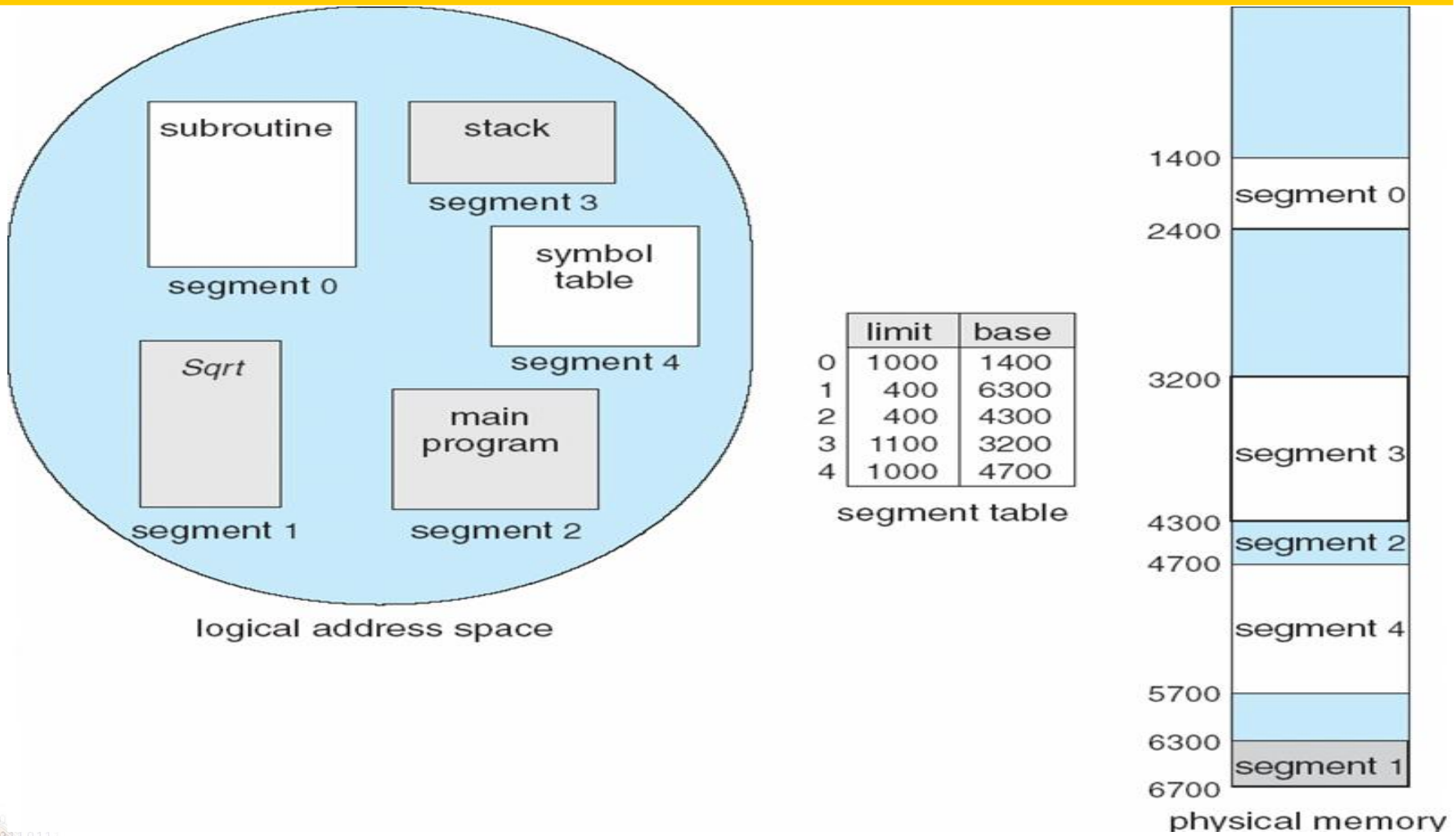
Segmentation Architecture

- **Segment table** – maps two-dimensional addresses to 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**

Example of Segmentation



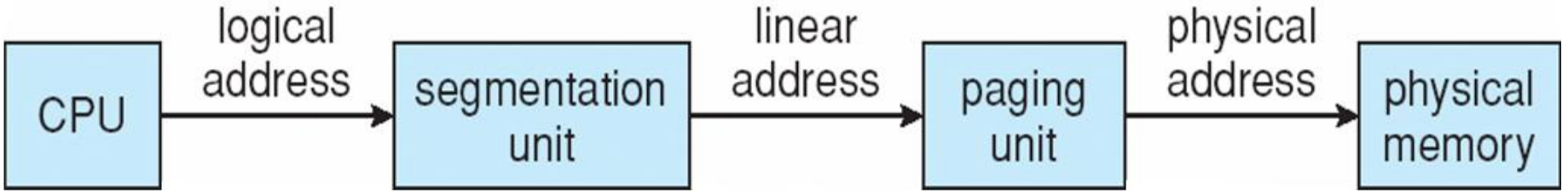
Segmentation Architecture (Cont.)

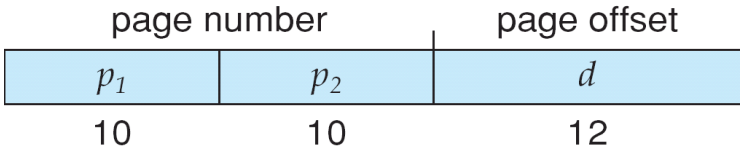
- Protection
  - With each entry in segment table associate:
    - ✓ validation bit = 0 ⇒ illegal segment
    - ✓ read/write/execute privileges
- Protection bits associated with segments; code sharing occurs at segment level
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem
  - Paging can be used to store segments….

Supports both segmentation and segmentation with paging

EXAMPLE: THE INTEL PENTIUM

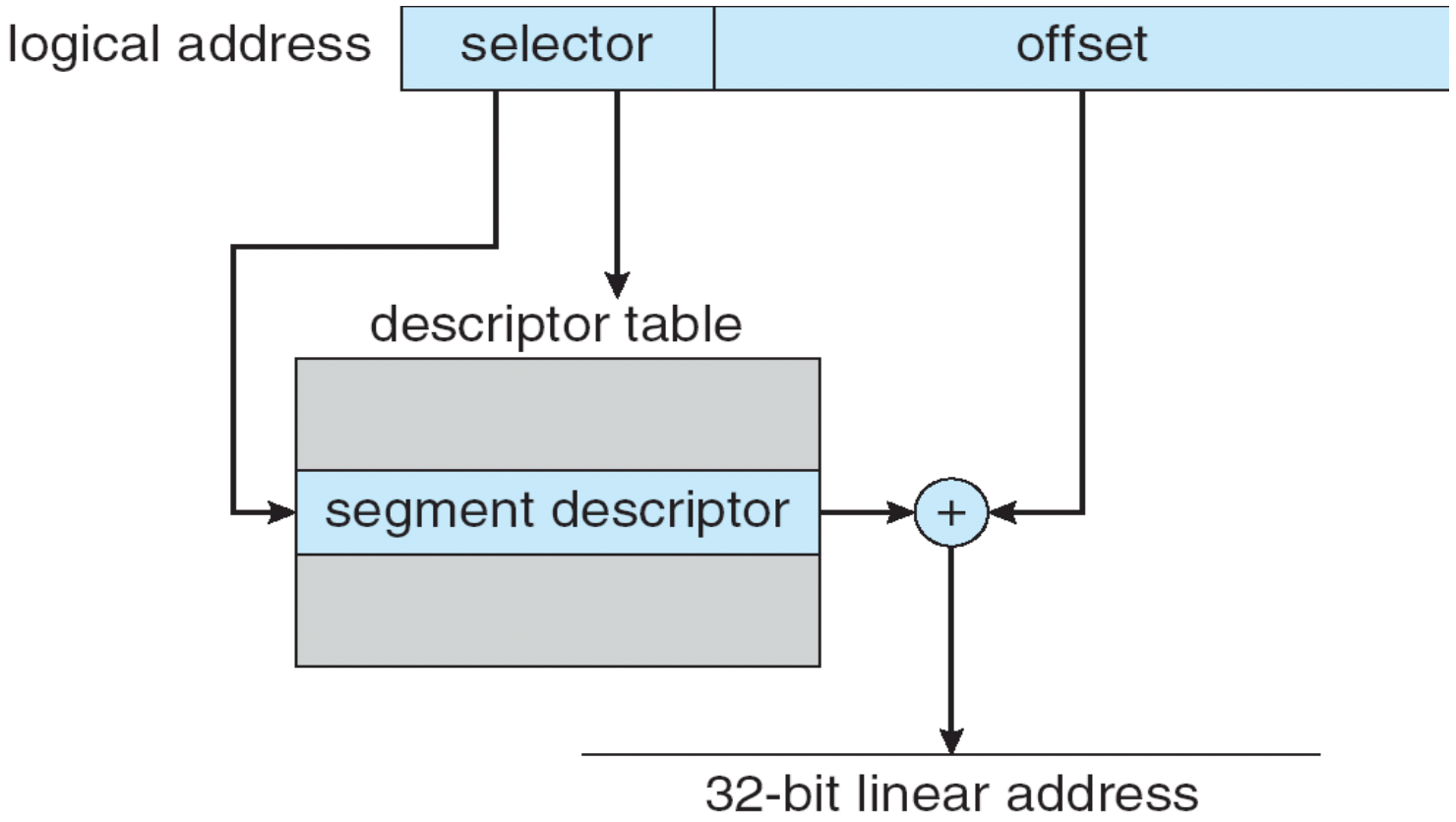
Logical to Physical Address Translation in Pentium



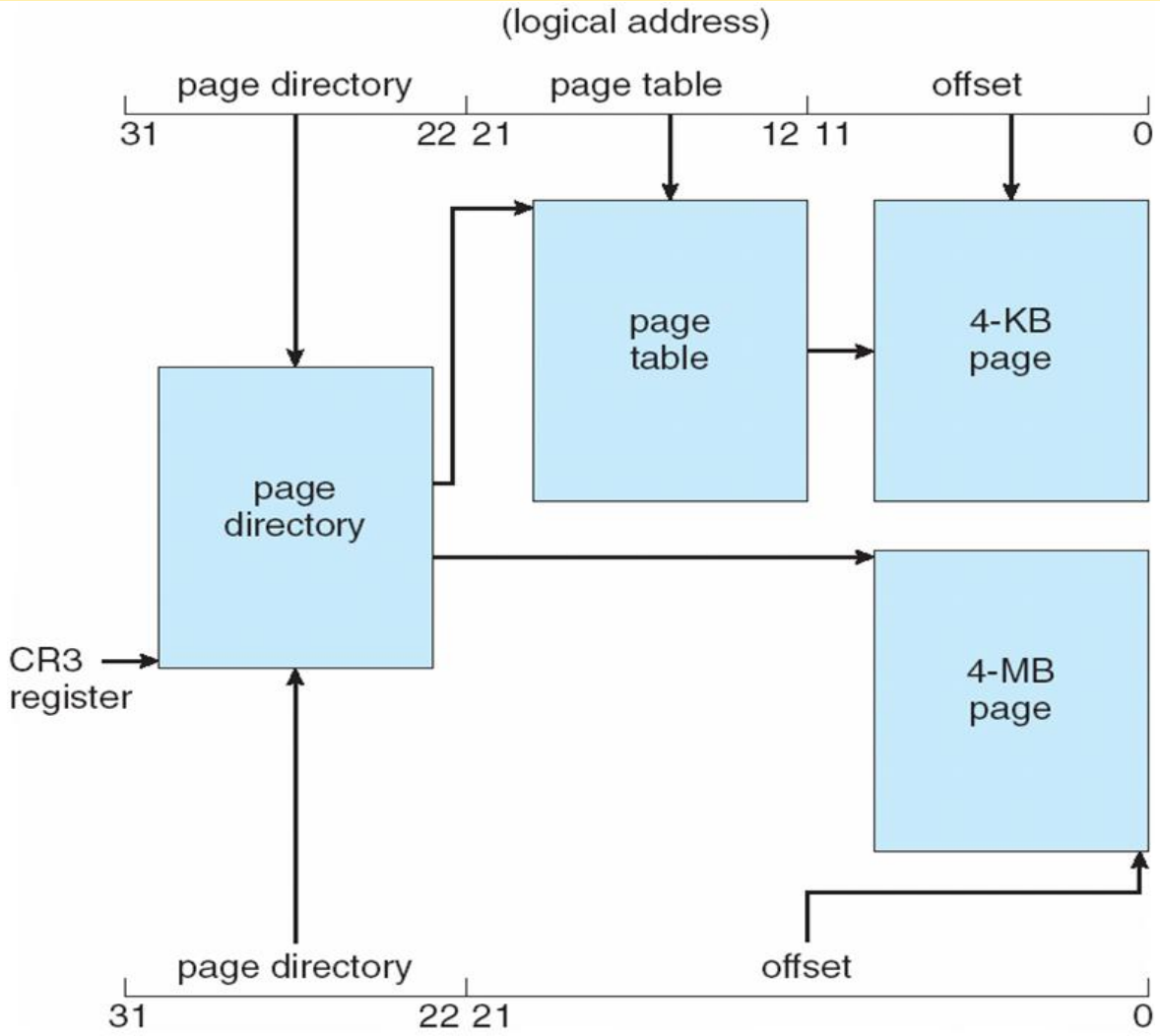


- CPU generates logical address
  - Given to segmentation unit
    - ✓ Which produces linear addresses
  - Linear address given to paging unit
    - ✓ Which generates physical address in main memory
    - ✓ Paging units form equivalent of MMU

Intel Pentium Segmentation

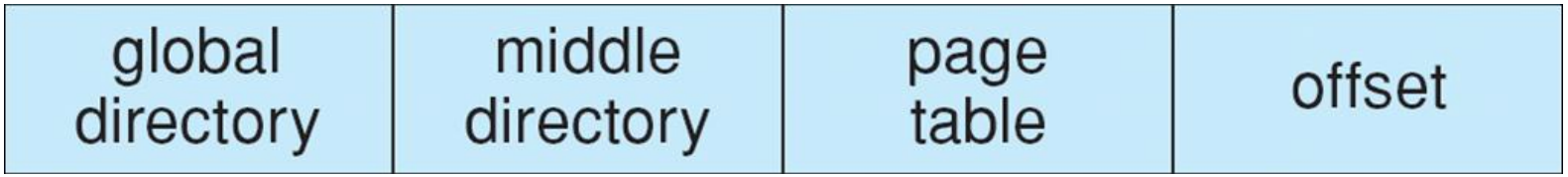


Pentium Paging Architecture



Linear Address in Linux

Broken into four parts:



Three-level Paging in Linux