

ECE437/CS481

M07A: MEMORY MANAGEMENT

CHAPTER 8.1-8.6

Xiang Sun

The University of New Mexico

A decorative blue wavy line that spans the width of the slide, starting with a small peak on the left, dipping into a valley in the center, and then rising to a small peak on the right before continuing as a straight line to the edge.

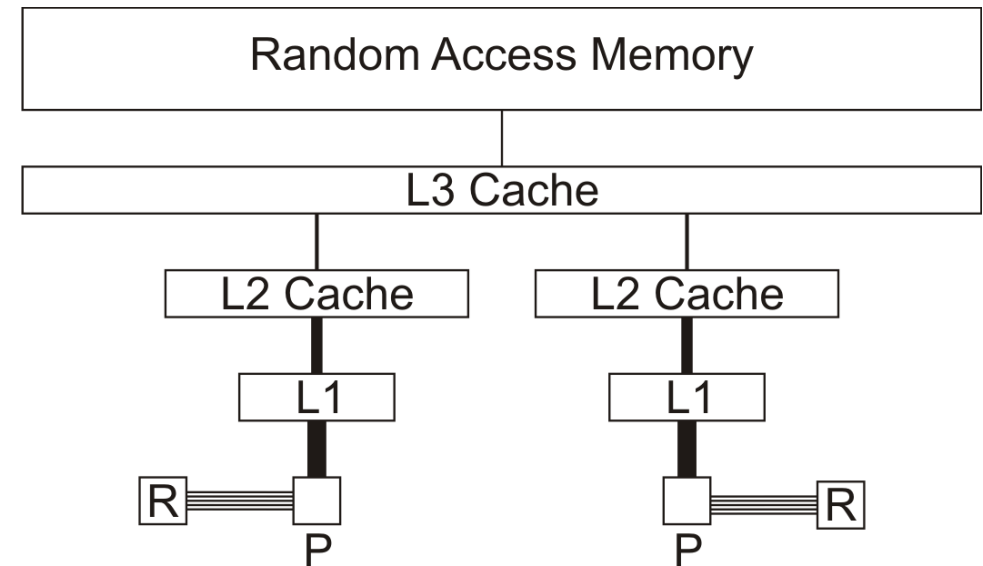
Goals and Tools of Memory Management

- ❑ Allocate memory resources among competing processes
 - maximizing memory utilization and system throughput.
- ❑ Provide isolation between processes
 - Addressability and protection: orthogonal
- ❑ Convenient abstraction for programming
 - and compilers, etc.
- ❑ Tool
 - Base and limit registers
 - Swapping
 - Segmentation
 - Paging, page tables and TLB
 - Virtual memory

Memory, cache, and CPU

□ Architecture

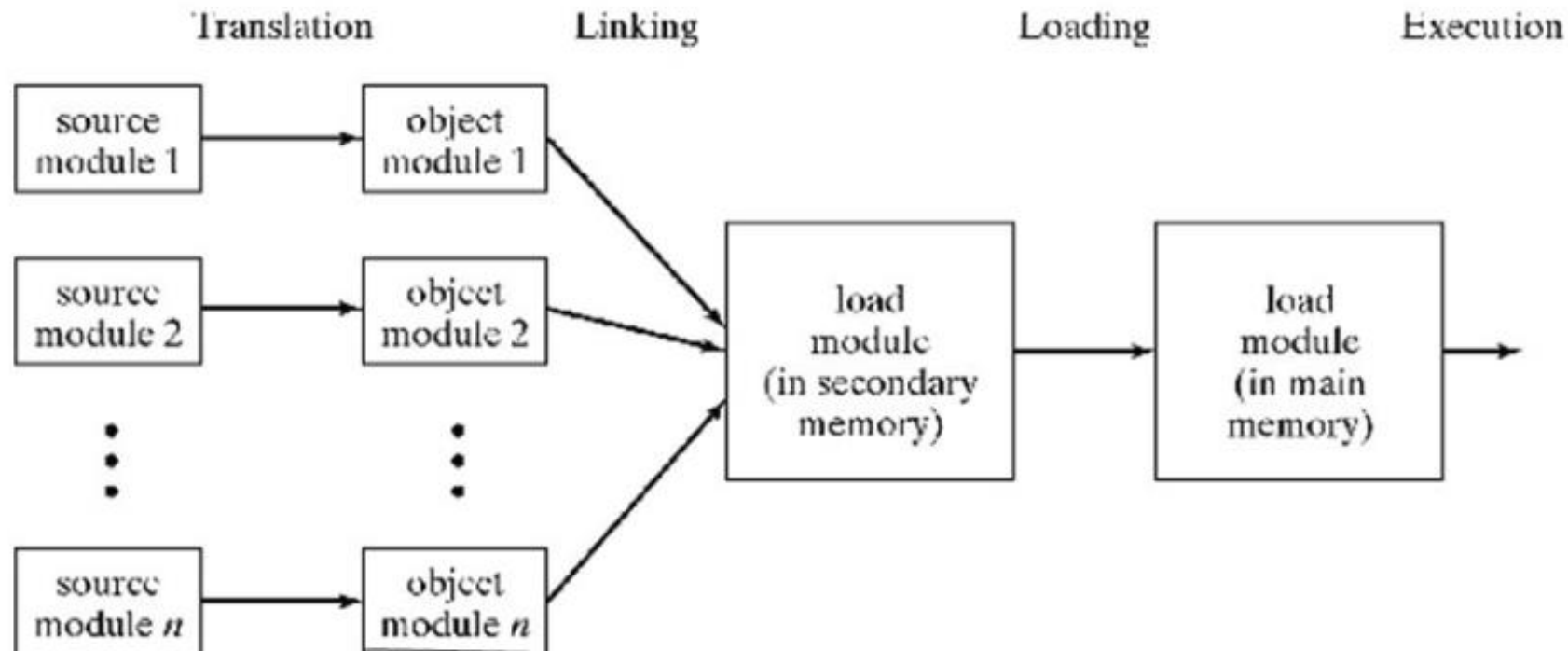
- To run a process, programs will be read from 2nd storage (disks) into memory.
- During the process lifetime, both instructions and data need moved between memory and CPUs.
- CPU can read/write
 - ✓ Registers--within a CPU clock
 - ✓ Memory--taking many cycles—**memory stall**
 - ✓ Cache--between main memory and CPU registers



Preparing Program for Execution

□ Program Transformations

- Translation (Compilation) of program
- Linking of program with other object modules needed for its execution
- Loading of the modules in the memory for purpose of execution



Preparing Program for Execution

❑ A sample code segment

Source code in the text editor

```
...
static int gVar;
...
int proc_a(int arg){
    ...
    gVar = 7;
    put_record(gVar);
    ...
}
```

Relocatable file in secondary memory

Code Segment		Code Segment	
Relocate Address	Generated Code	Relocate Address	Generated Code
0000	(Other module)	...	
...		3136	[Space for gVar variable]
1008	entry proc_a	...	
...		4000	(last location in the data segment)
1220	load 7, %rax		
1224	store %rax, 3136		
1228	push %rbx		
1232	call 2334		
...			
1399	(End of proc_a)		
...	(Other modules)		
2334	entry put_record		
...			
2399	(End of put_record)		
...			
2999	(Last location in the code segment)		

Load module in main module
(The Program Loaded at Location 4000)

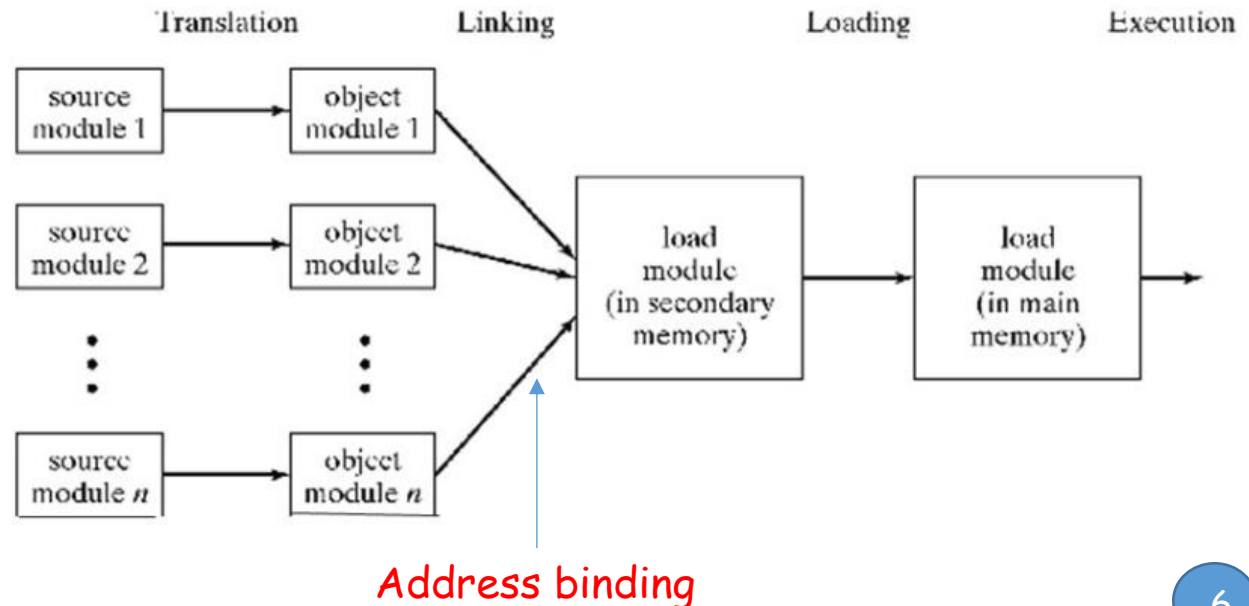
Absolute Address	Generated Code
0000	(Other process's programs)
4000	(Other modules)
...	
4000	(Other modules)
...	
5008	entry proc_a
...	
5220	load 7, %rax
5224	store %rax, 7136
5228	push %rbx
5232	call 6334
...	
6399	(End of proc_a)
...	(Other modules)
6334	entry put_record
...	
6399	(End of put_record)
...	
6999	(Last location in the code segment)
7000	(first location in the data segment)
...	
7136	(Space for gVar variable)
...	
8000	(last location in the data segment)

- Relocate addr: the type of address that needs to be relocated during the linking procedure.
- Logical/Virtual addr: the type of address within a process after linking.
- Physical addr: the type of address in the physical memory.

Address Binding

□ Address binding

- Binding the logical address of codes and data to physical memory address (i.e., absolute address)
- The address binding can happen at three different stages
 - ✓ **Compile time:** the type of address in the ELF file stored in the secondary memory is the **physical memory address**.
 - Compiler has to know the main memory location in advance.
 - Compiler has to recompile the codes if the location is changed.

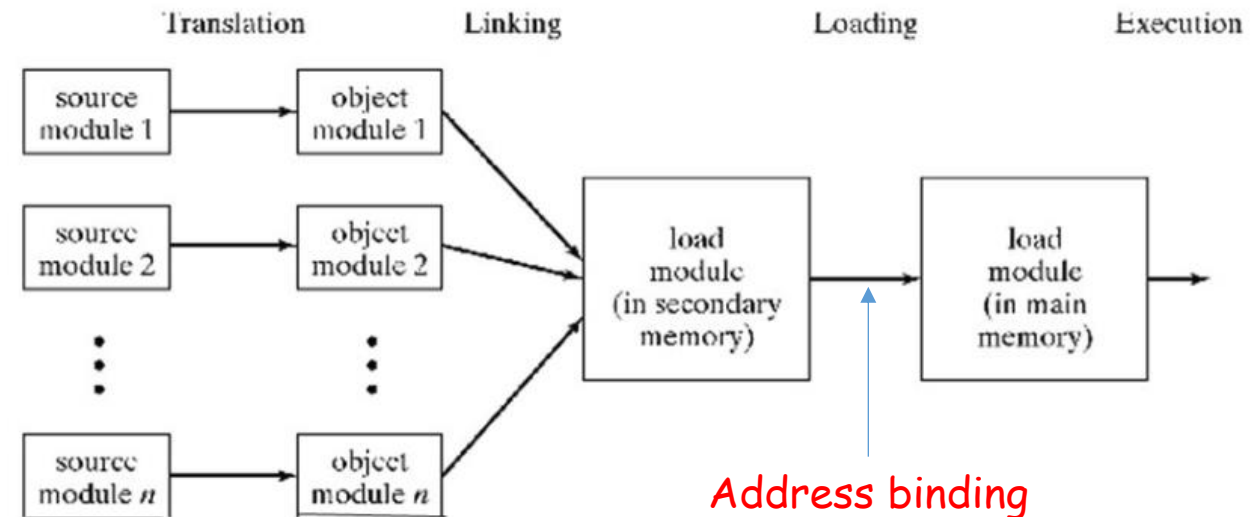


Address Binding

□ Address binding

- Binding the logical address of codes and data to physical memory address (i.e., absolute address)
- The address binding can happen at three different stages
 - ✓ **Loading time:** the type of address in the ELF file stored in the secondary memory is the **logical address**. But the type of address in the virtual memory layout is the **physical memory address**.

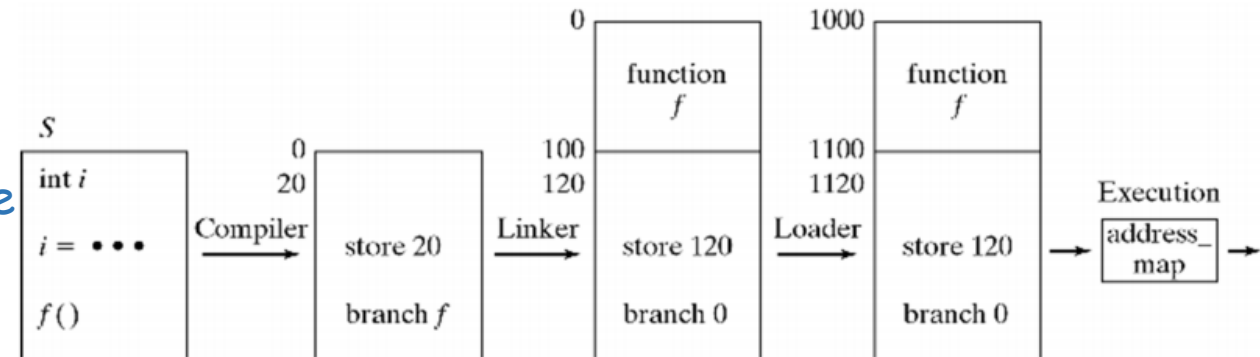
- Compiler does not know the main memory location.
- The address binding happens when the program is loaded into the main memory.



Address Binding

□ Address binding

- Binding the logical address of codes and data to physical memory address (i.e., absolute address)
- The address binding can happen at three different stages
 - ✓ **Running time:** the type of address in the ELF file stored in the secondary memory and in the virtual memory layout is the **logical address**. But the **logical address** will be translated into the **physical memory address** during the execution.
 - Compiler does not know the main memory location.
 - The address binding happens when the process is executed.

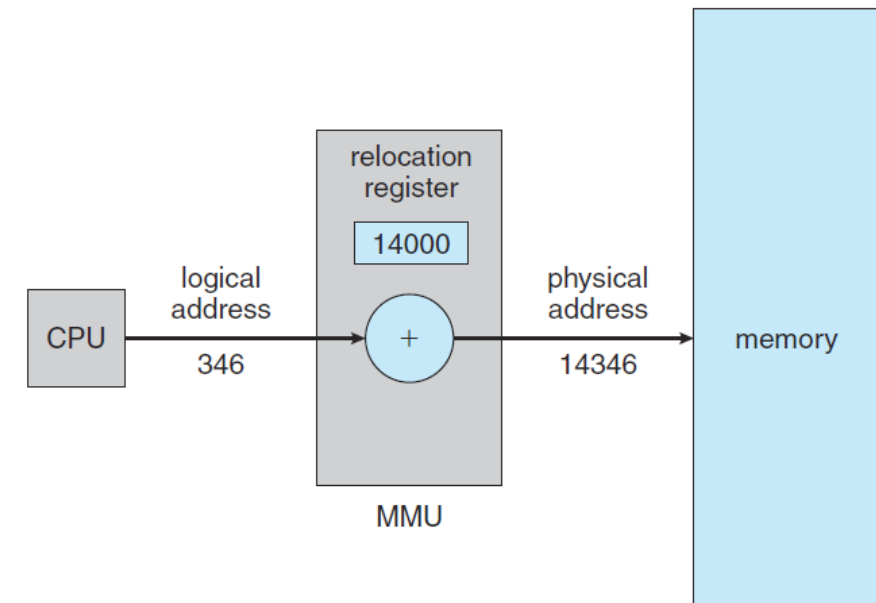
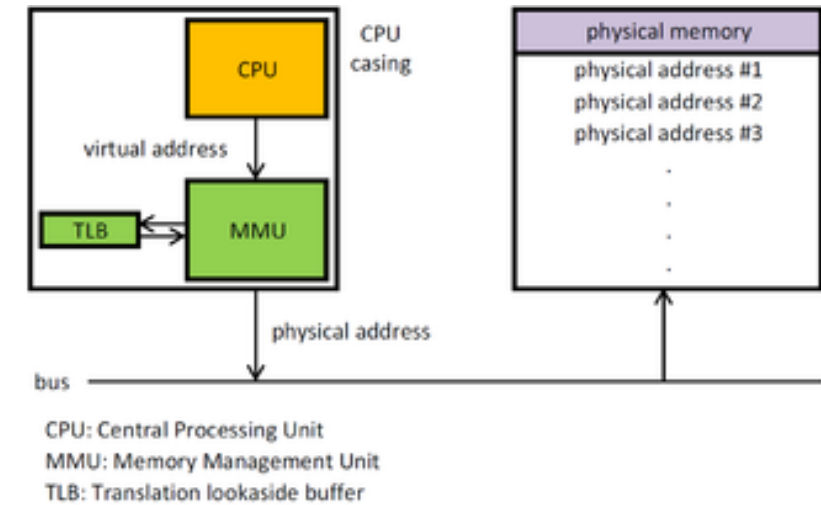


Address Binding

□ Running time address binding

- Running time mapping from logical address space into a separate physical address space
- The mapping is achieved by Memory Management Unit (MMU, hardware support).
 - ✓ With MMU, the value in the relocation register is added to every address generated by a process at the time it is sent to memory

$$\text{Physical address} = \text{Logical address} + \text{relocation register}$$



Address Binding

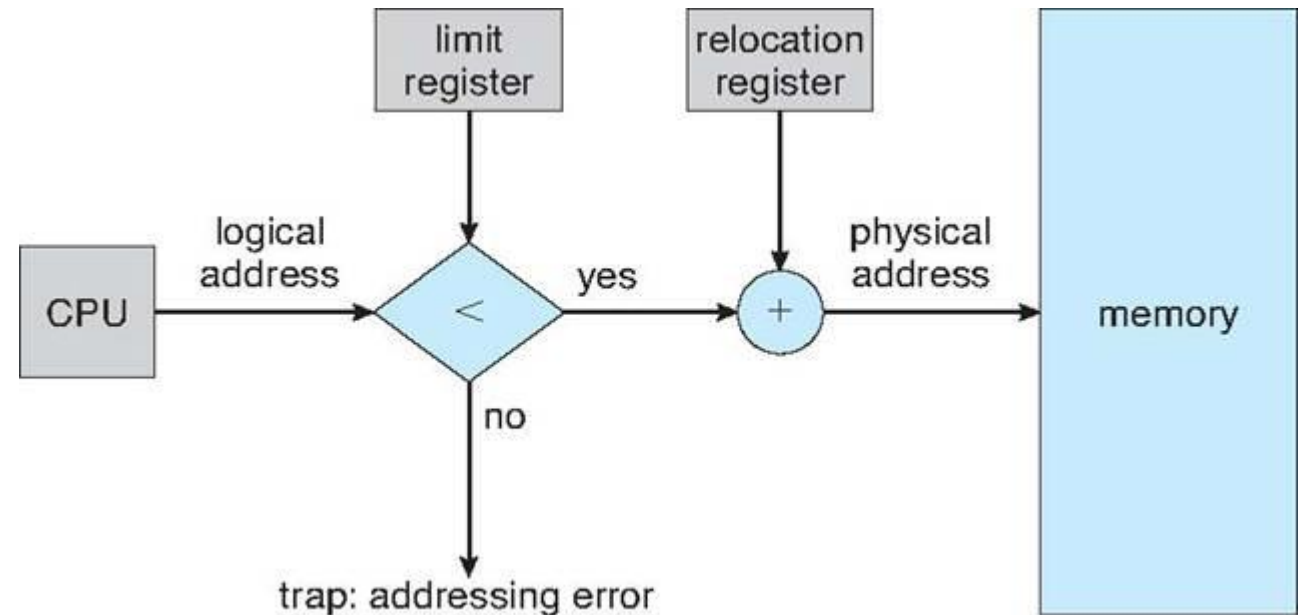
□ Running time address binding

- In order to avoid the process in accessing other processes' memory spaces, the relocation/base register and the limit register are used.
 - ✓ Limit register contains range of logical addresses - each logical address must be less than the limit register

□ Is logical address beyond the limit register?

- If yes, "trap" to deal with memory access errors
- Otherwise, go to physical address

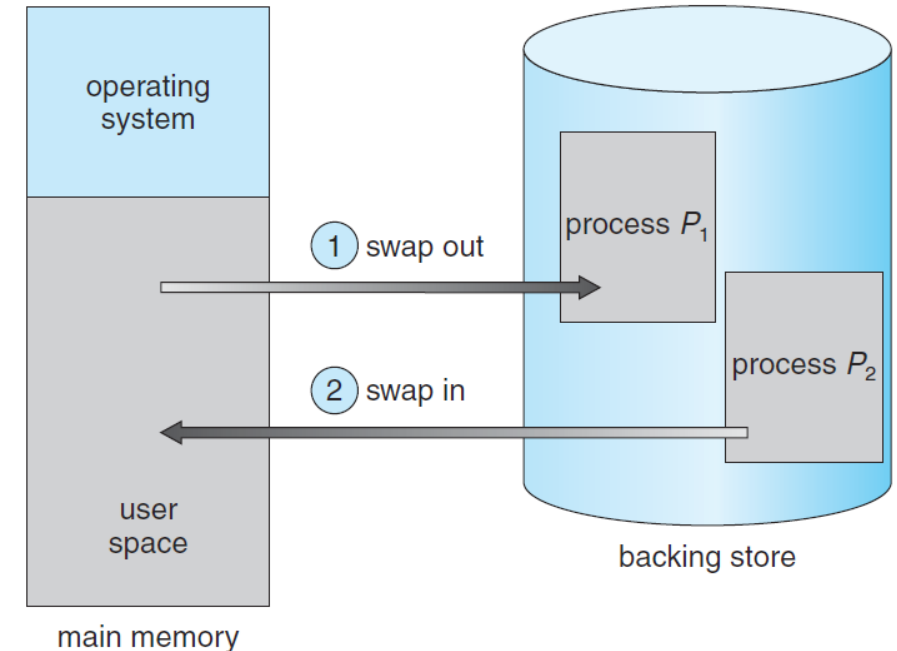
Physical address =
Logical address + base/relocation register



Swapping

❑ Swapping (Recall)

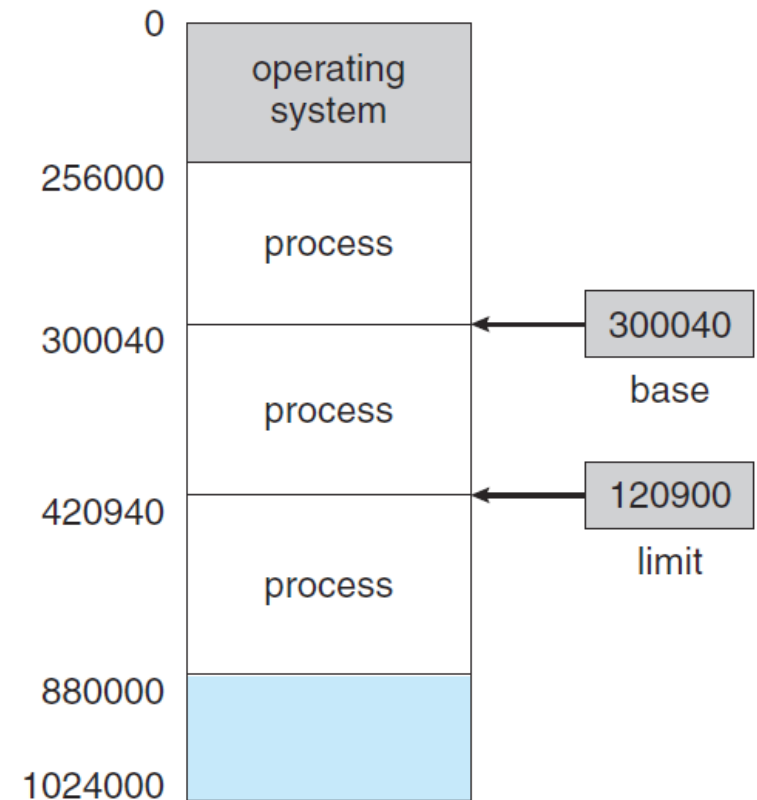
- A process can be swapped temporarily out of memory to the backing store (e.g., hard drives), and then brought back into memory for continued execution
 - ✓ **Swap out**: memory-to-backing store
 - ✓ **Swap in**: backing store-to-memory
- The short-term CPU scheduler (dispatcher) decides which process is executed next. After the decision, it checks if the desired process is in the ready queue in the memory.
 - ✓ If yes, does nothing.
 - ✓ If no, checks if there is enough memory space for the desired process to be swapped in.
 - If yes, swaps in the process
 - If no, swaps out **a process** currently in memory and swaps in the desired process



Memory Allocation

□ Memory allocation

- Main memory is divided into two **partitions**: Resident operating system (usually held in low memory) and user processes (held in high memory).
- Main memory has limited resource, and thus must be efficiently allocated.
- **Contiguous memory allocation** is one method.
 - ✓ Each process is contained in a partition with single **contiguous** section of memory
 - ✓ Memory accessing is protected by relocation + limit registers
 - ✓ MMU maps logical address dynamically



Memory Allocation

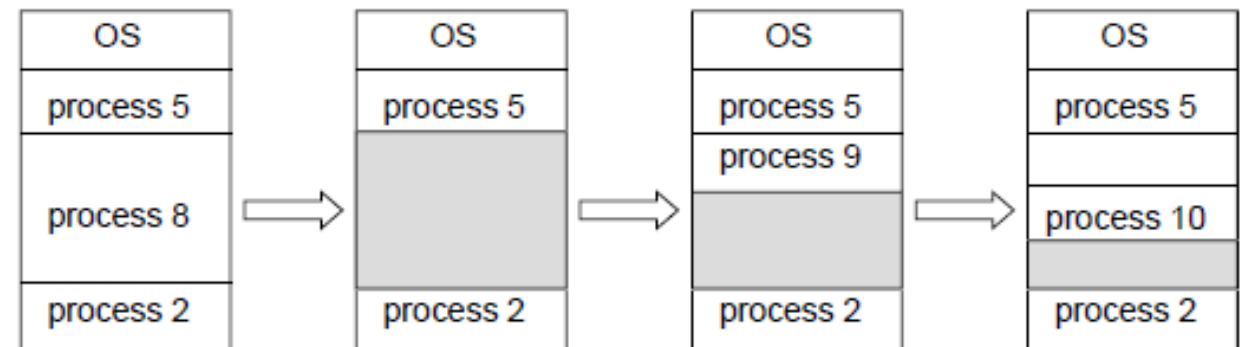
❑ Contiguous allocation

- Contiguous allocation isn't allocating memory spaces to processes sequentially.

- ✓ Processes may terminate
- ✓ Processes may be swapped in/out
- ✓ New processes may arrive

- Main memory space may be separated into small **holes**.

- ✓ Hole - a 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
- ✓ OS maintains information about:
 - Allocated memory blocks
 - Free memory blocks (holes), sorted by memory address or sizes



Memory Allocation

- ❑ How to satisfy a process requesting of n units memory space from a list of free holes
 - First fit
 - ✓ allocate the **first** hole that is big enough
 - ✓ may incur many small holes
 - Best fit
 - ✓ allocate the **smallest** hole that is big enough
 - ✓ may search entire free list
 - ✓ produces the smallest leftover hole
 - Worst fit
 - ✓ allocate the **largest** hole
 - ✓ must search entire free list
 - ✓ produces the largest leftover hole

Memory Allocation

❑ Contiguous allocation incurs external fragmentation

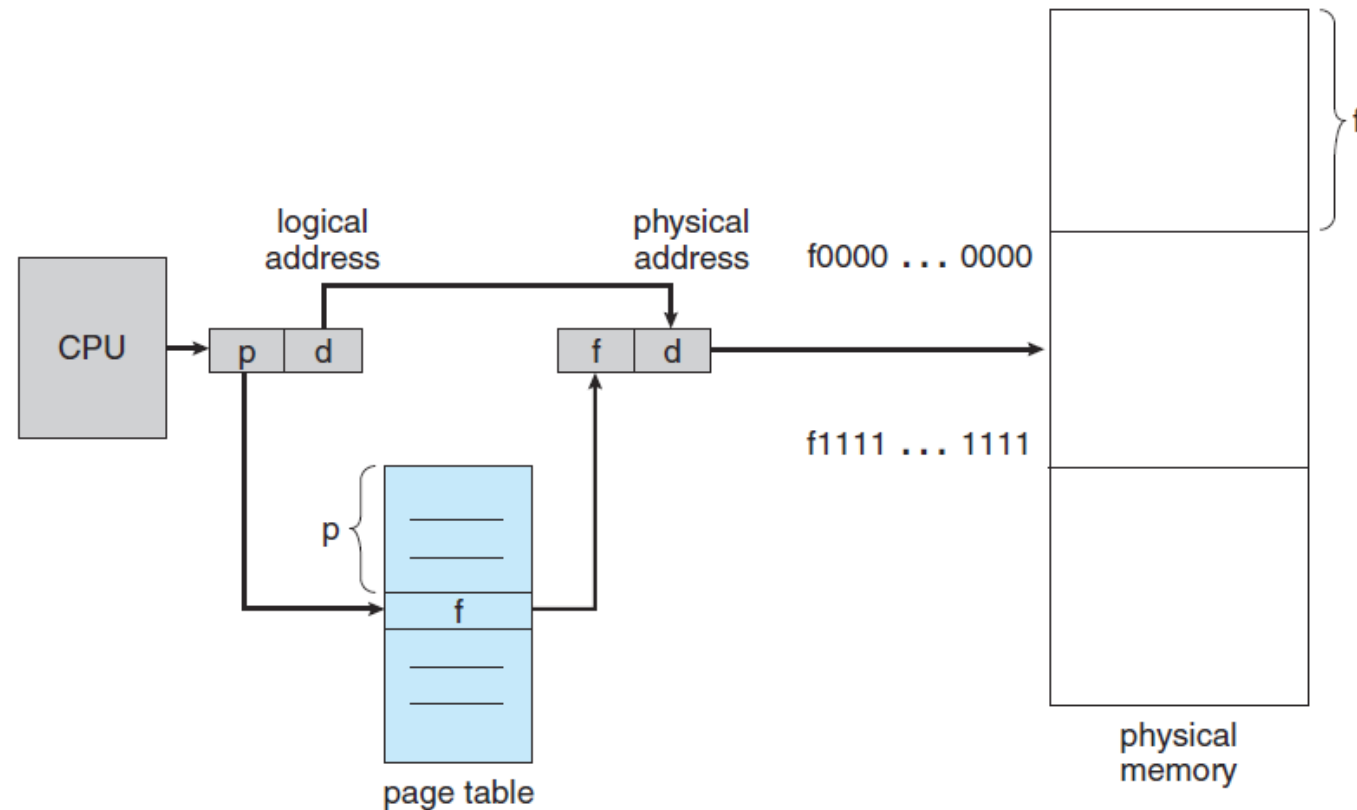
- External fragmentation: memory free space can satisfy a request, but it is not contiguous.
- Solution:
 - a) Reduce external fragmentation by **compaction**
 - ✓ move allocated partitions (blocks) to place all holes together
 - ✓ compaction is possible only if relocation is dynamic (running time address binding)
 - b) Permit non-contiguous allocation of logical address space---Paging
 - ✓ Divide physical memory into fixed-sized blocks called **frames** (the size is power of 2, between 512 (2^9) bytes and 1GB (2^{30} B))
 - ✓ Divide logical memory into blocks of same size called **pages**
 - ✓ Keep track of all free frames.
 - ✓ To run a program of n-page size, need to find n free frames in the memory and load program
 - ✓ Set up a **page table** to **translate logical addresses to physical addresses**

Paging

□ Paging

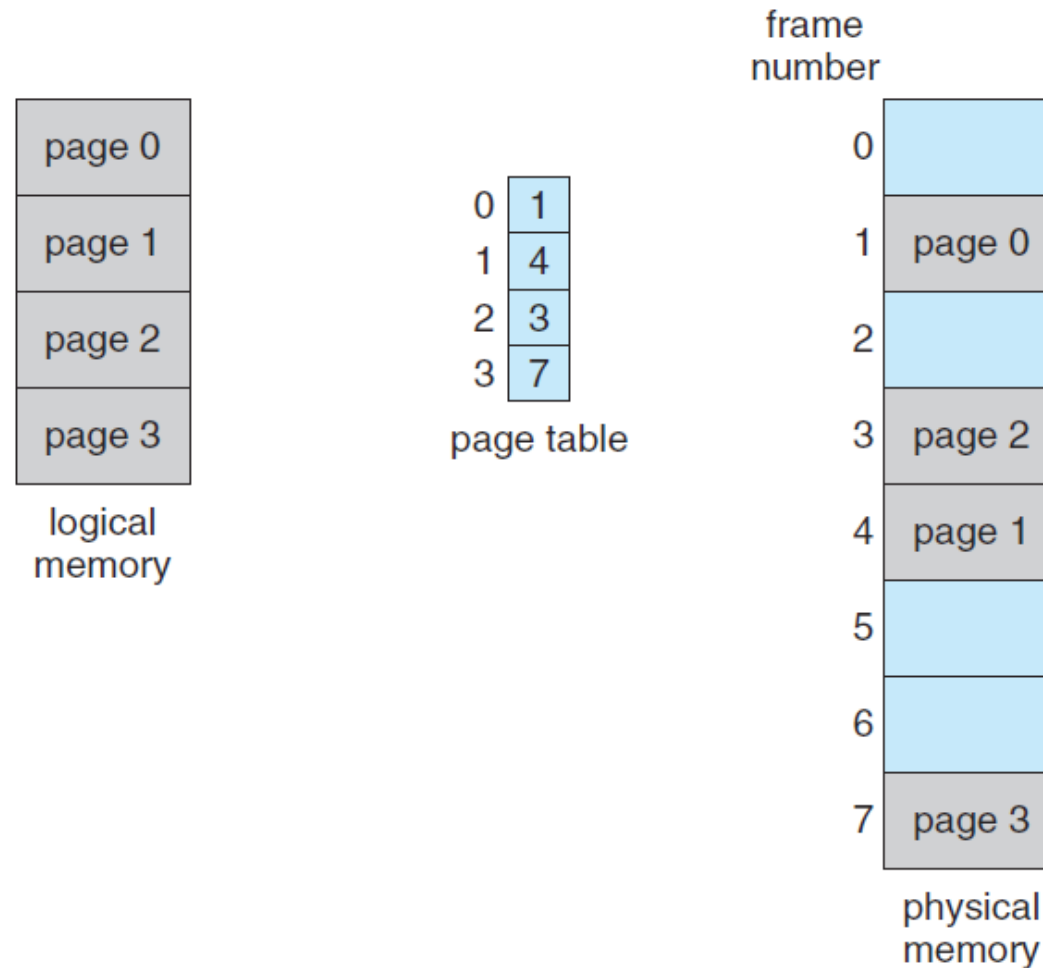
➤ Address generated by CPU is divided into:

- ✓ Page number (p) - used as an **index** into a **page table** which indicates base address of each frame in physical memory
 - Page table: mapping logical to physical addresses
- ✓ Page offset (d) - combined with base address to define the physical memory address



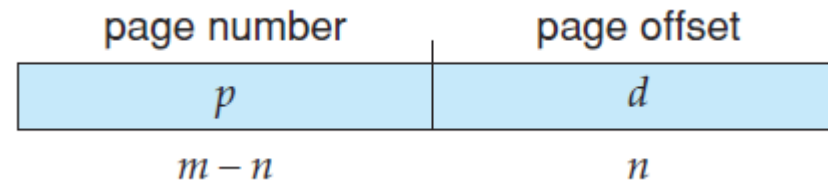
□ Address Translation Scheme in Paging

➤ Example: logical page number → physical frame number



□ Address Translation Scheme in Paging

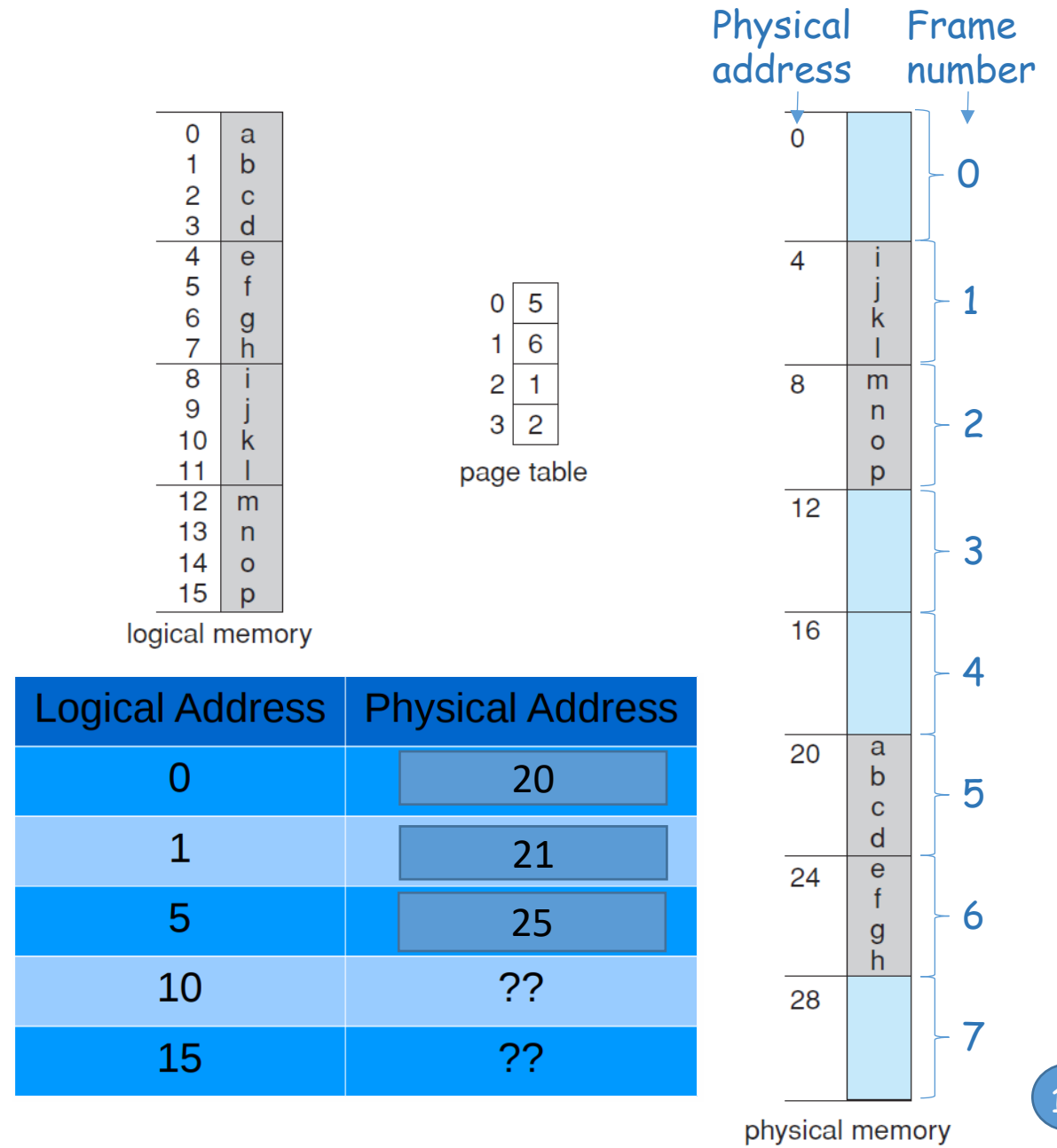
- The page size/frame size is defined by the hardware (2^9 -- 2^{30} B).
- If the size of the memory address space is 2^m and the size of a page is 2^n , then the high-order $m-n$ bits of a logical address designate the page number, and the n low-order bits designate the page offset.



Paging

□ Address Translation Scheme in Paging

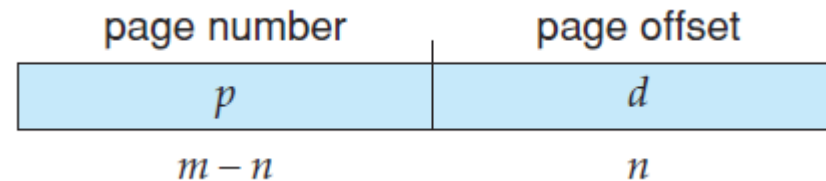
- If $n=2$ and $m=5$, how a logical memory is mapped into a physical memory?
- When we use a paging scheme, we have no external fragmentation--any free frame can be allocated to a process that needs it.
- Question: A 32-bit computer system uses 32-bit to represent a memory address, and the frame size is 4KB, what is the maximum physical memory address that can be addressed?



Paging

❑ Paging may incur **internal fragmentation**

- **Internal fragmentation:** allocated memory may be slightly larger than requested memory.
- In the paging system, frames are considered as smallest units allocated to processes. If the memory requirements of a process do not happen to coincide with page boundaries, the last frame allocated may not be completely full.
 - ✓ if page size is 2,048 bytes and a process requires 71,681 bytes memory, how many frames are needed?
 - ✓ $71,681 / 2,048 = 35 \text{ frames} + 1 \text{ bytes}$ → the system would allocate 36 frames to the process and the last frame only utilize 1 byte (low memory utilization).
- In order to reduce internal fragmentation loss, we may choose smaller page size; however, a smaller page size (n) incurs a larger size of page table ($m-n$).



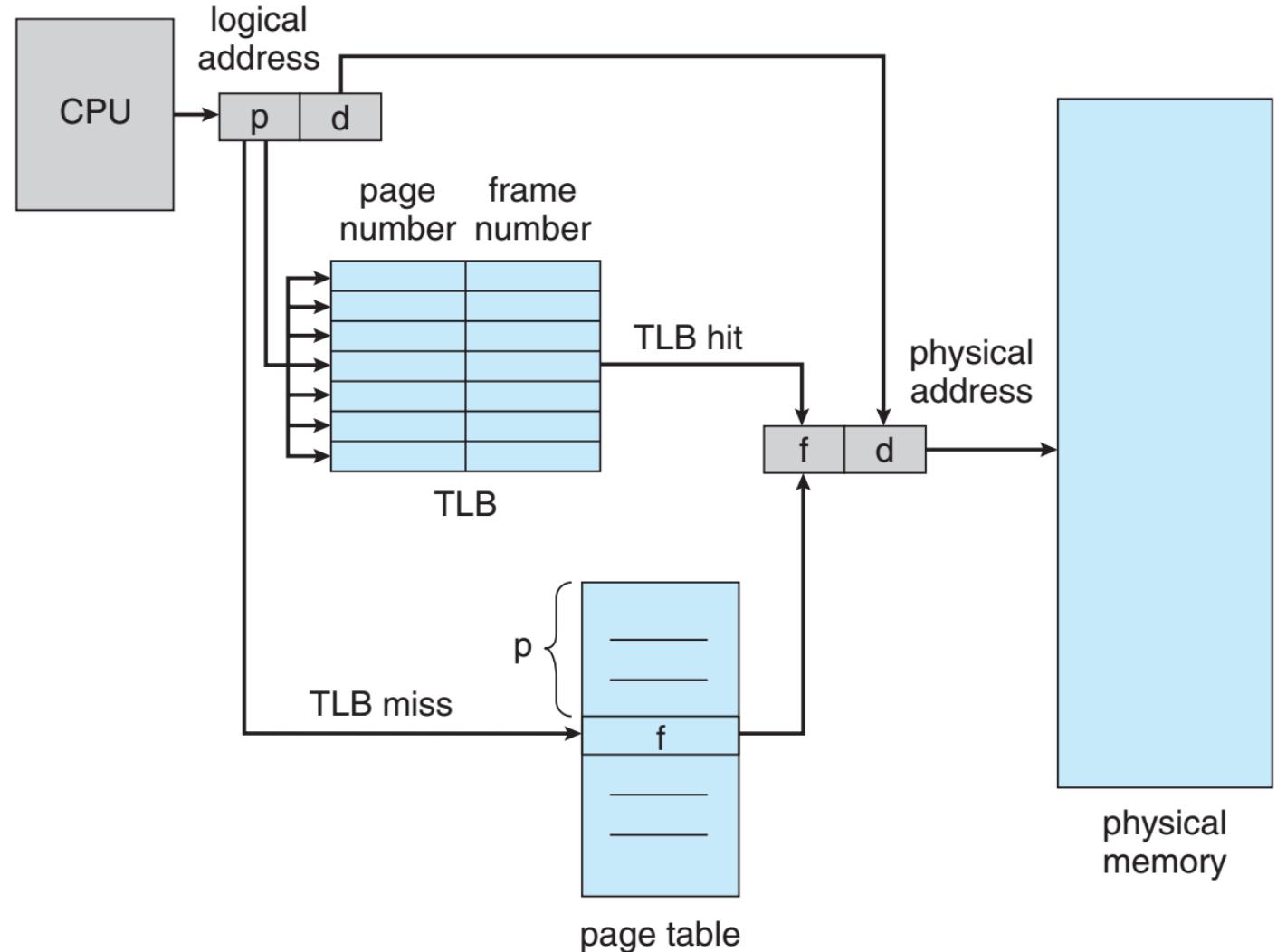
□ Page table implementation

- Each process has its own page table in the main memory.
- The **Page-table base register (PTBR)** is used to point to the page table of the running process.
 - ✓ The PTBR is changed on each context switch.
- Every load/store requires two memory accesses
 - ✓ one for accessing page table and one for accessing the data/instruction.
 - ✓ Thus, memory access is slowed by a factor of 2, which is not efficient.
- Solution: use a special, small, fast-lookup hardware cache, **called a translation look-aside buffer (TLB)**, to **hold popular page table entries**.
 - ✓ access to TLB should be very fast

Paging

□ Page table implementation with TLB

- ✓ The TLB contains only a few page-table entries (typically between 32 and 1,024 entries).
- ✓ When a logical address is generated by the CPU, its page number is presented to the TLB.
 - If the page number is found (known as a **TLB hit**), its frame number is immediately available and is used to access memory
 - If the page number is not in the TLB (known as a **TLB miss**), a memory reference to the page table must be made.



❑ Page table implementation with TLB

- Problem: each process has its own page table; once a context switch happens, the OS changes PTBR pointer. How about page table entries cached in TLB?
 - ✓ TLB may include old entries that contain valid virtual addresses but have incorrect or invalid physical addresses left over from the previous process.

➤ Solution:

- TLB must be **flushed/erased** in each context switch
- Each entry in TLB is associated with an **address-space identifier (ASID)**
 - ✓ ASID uniquely identifies each process to provide address-space protection for that process
 - ✓ When the TLB attempts to resolve virtual page numbers, it ensures that the ASID for the currently running process matches the ASID associated with the virtual page. If the ASIDs do not match, the attempt is treated as a TLB miss.

TLB		
ASID	Page number	Frame number

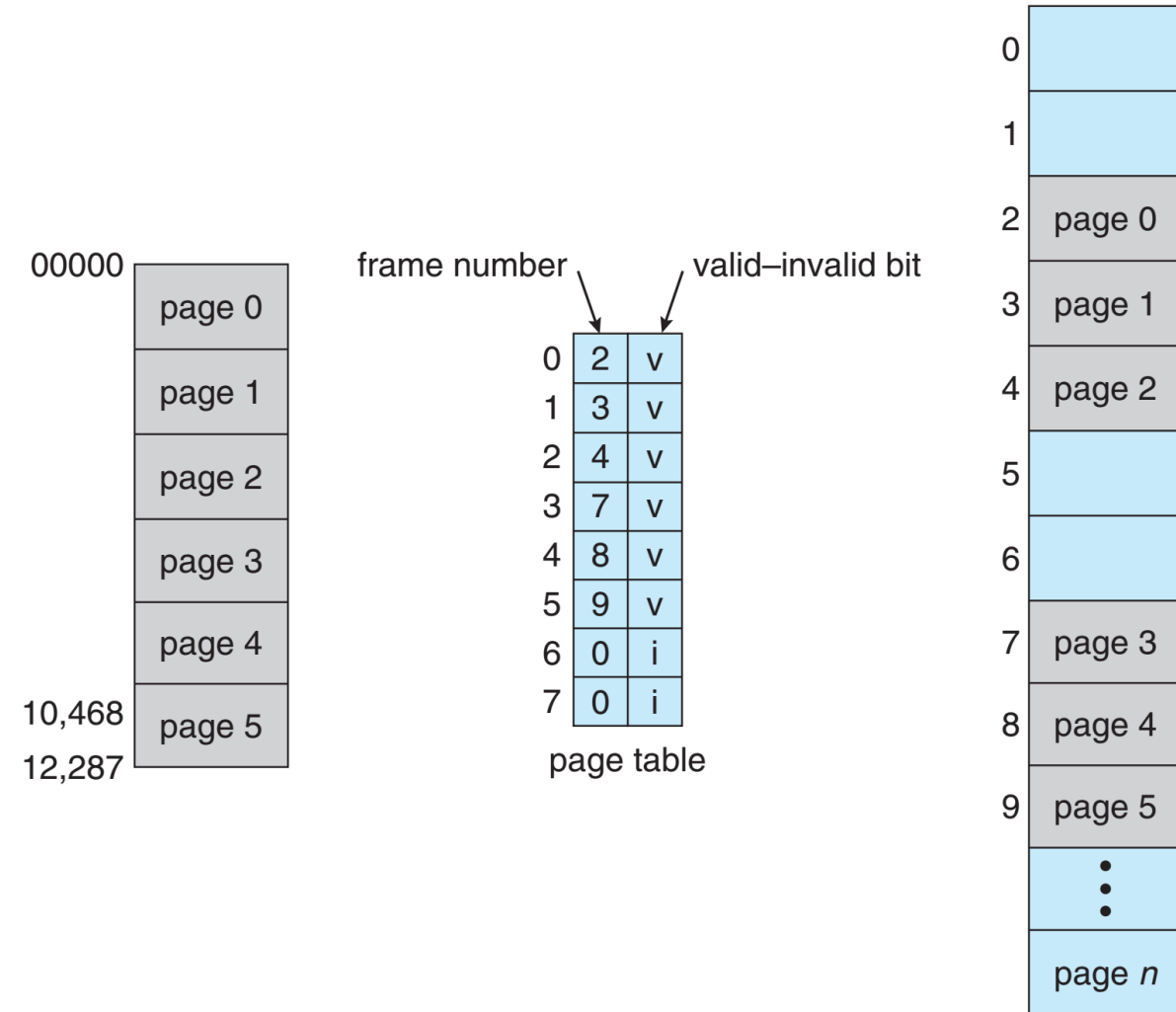
❑ Memory protection

- Memory protection implemented by associating **protection bits** with each frame in the page table to indicate the frame is
 - ✓ read-only or
 - ✓ read-write access allowed
 - ✓ execute only
 - ✓ etc...
- Also, **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

❑ Memory protection--Example

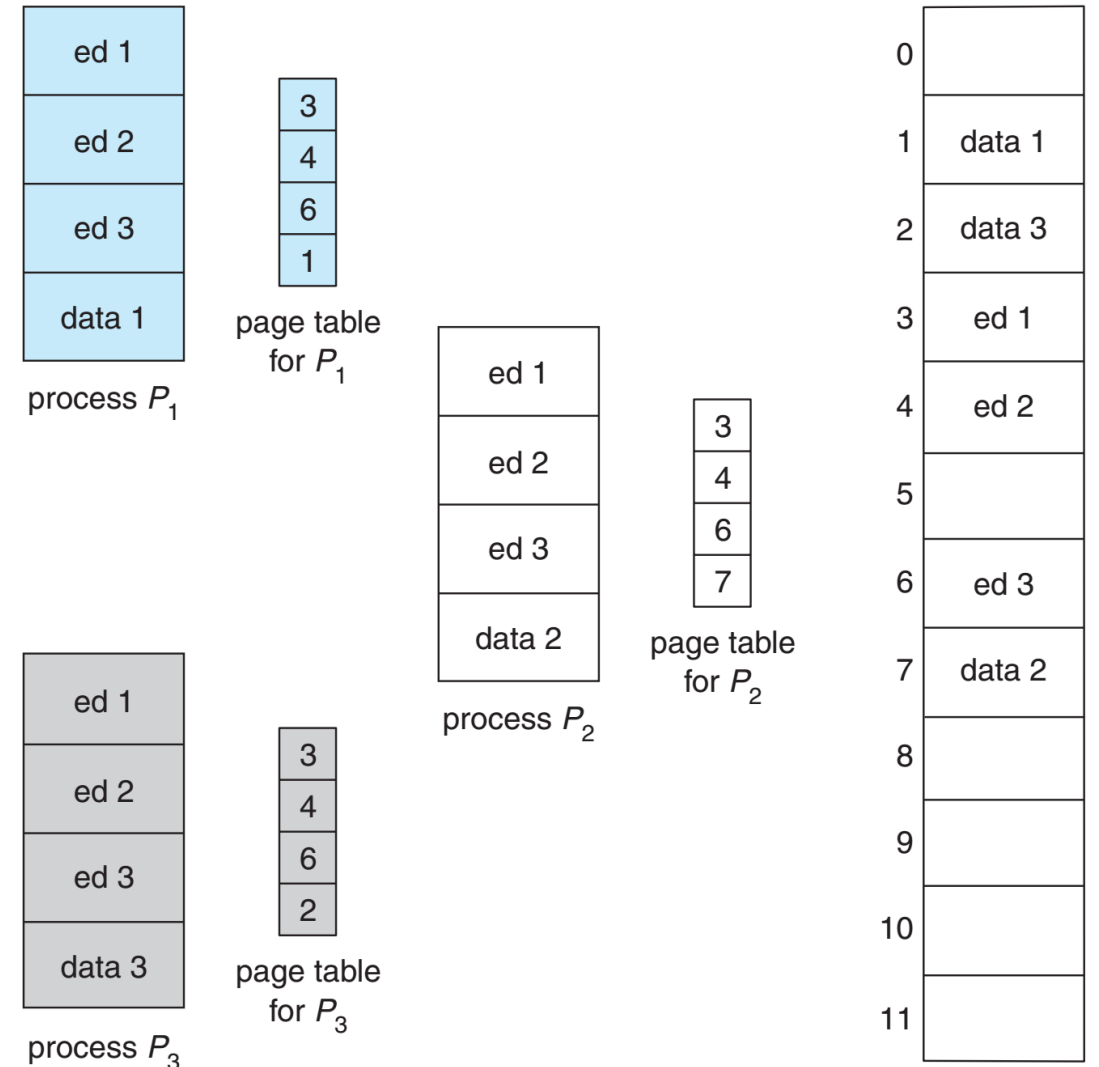
- A system has 14-bit address space
- A process uses addresses 0 - 10,468
- Page size is 2KB
- Accessing addresses on pages 6 and 7 is invalid
- Some systems use a **page-table length register (PTLR)** to indicate the size of the page table.
 - ✓ Check the value of **PTLR** to verify that the address is in the valid range for the process.



Paging

□ Share pages

- Paging allows the possibility of easily sharing code
- One copy of read-only (reentrant) code shared among processes
 - ✓ E.g., text editors, compilers, window systems
- Each process has private data
- Share code can significantly reduce memory usage.

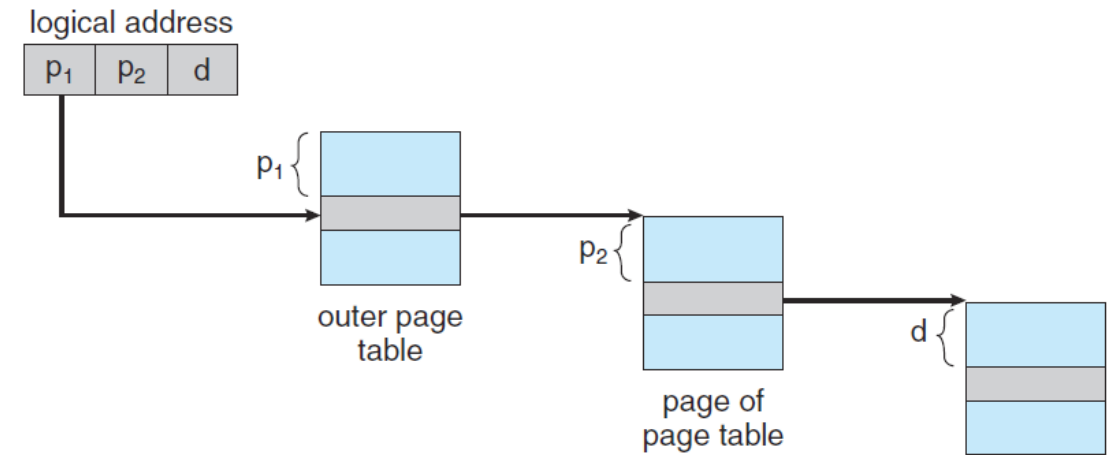
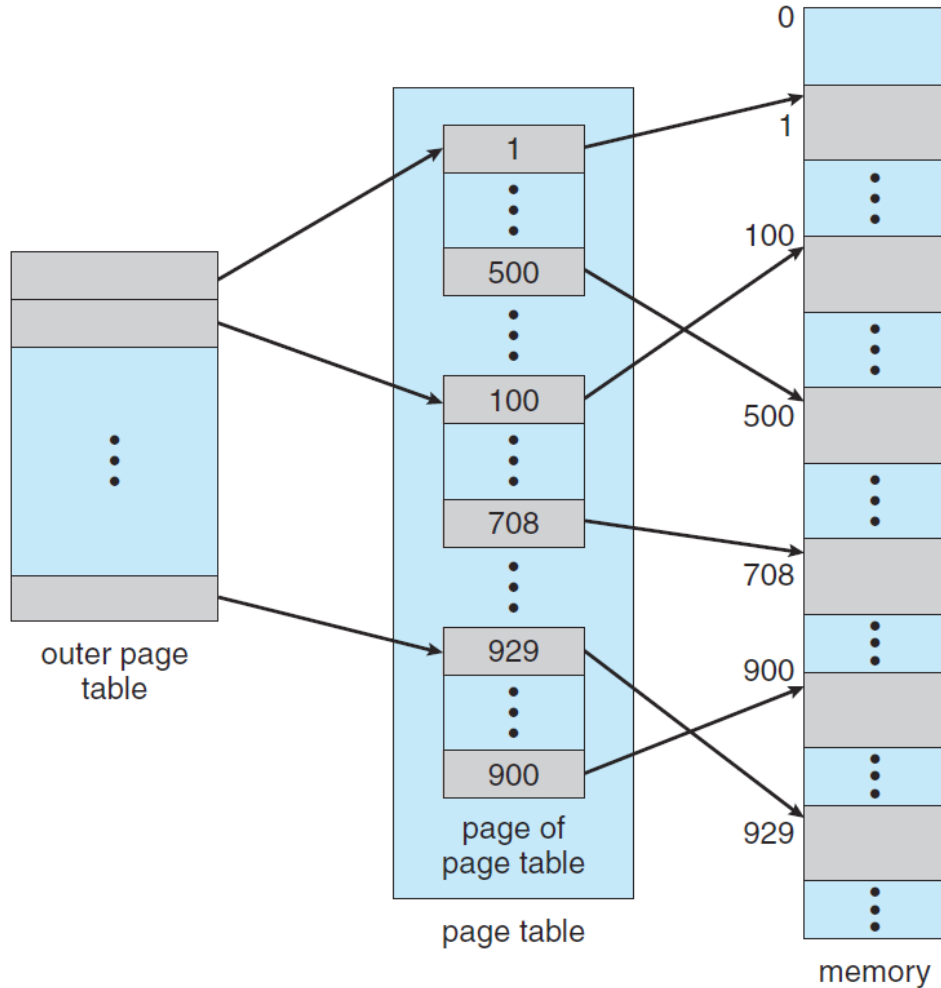


❑ Hierarchical Page Tables

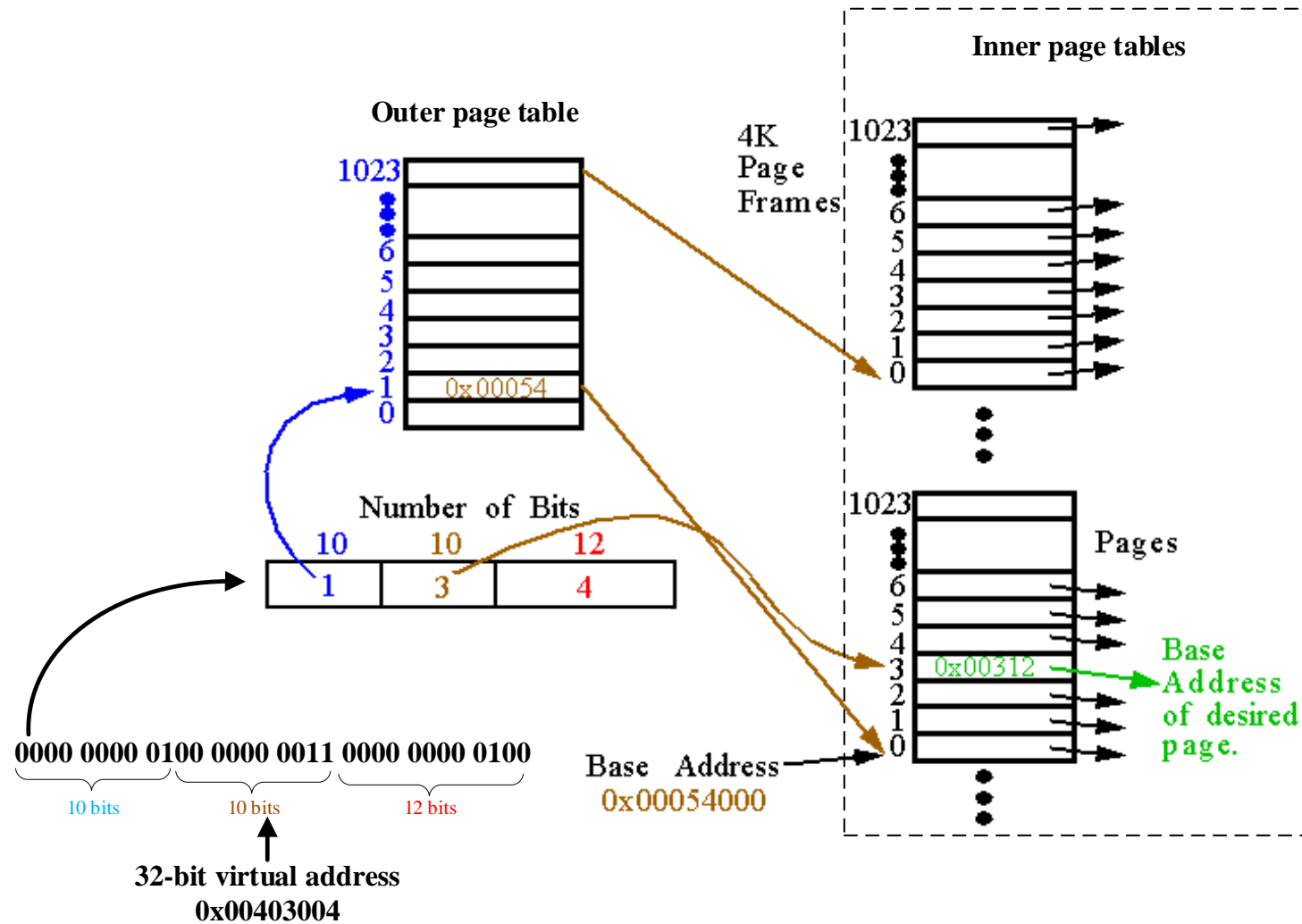
- The size of a page table is large
 - ✓ If a system with a 32-bit logical address space and the page size is 4 KB (2^{12}), then a page table may consist of up to $2^{32}/2^{12}=2^{20}$ entries.
 - ✓ If each entry needs 4 bytes, then the page table with 2^{20} entries needs 4 MB of physical space.
 - ✓ 4MB of memory space **per process** is a lot.
 - ✓ In addition, most of the processes only use a small portion of the address table (i.e., address space).
- Two-level page table (paged page-table)
 - ✓ One "Page number" column in the page table is further divided into two "Page number" columns, p1 and p2.
 - ✓ p1 - index of the entries in the outer page table
 - ✓ p2 - index of the entries in the inner page table

page number		page offset
p_1	p_2	d
10	10	12

□ Address-Translation Scheme for Two-Level Paging



□ Address-Translation Scheme for Two-Level Paging--example



❑ Two-level page table


➤ Could two-level page table reduce the address space?

- ✓ An outer page table has 2^{10} entries, each of which needs 4 bytes. Thus, the outer page table requires $4 \times 2^{10} = 4\text{KB}$
- ✓ There are 2^{10} inner page tables, each of which has 2^{10} entries. Thus, the inner page tables require $4 \times 2^{10} \times 2^{10} = 4\text{MB}$
- ✓ **Recall that:** single table page needs 4MB space.

➤ Two-level page table **can** reduce since most of inner page tables are NOT needed.

- ✓ Suppose that a process needs 4MB memory space- that is **only one inner page table is enough**.
- ✓ Thus, only one entry in the outer page table is marked as "**valid**" and associated to the specific inner page table. The rest of the entries in the outer page table are marked as "**invalid**" and are not associated to inner tables

Index	inner table number	valid/invalid bit
0	1	v
1	2	i
2	3	i
3	7	i
...



Index	frame number
0	11
1	12
2	13
3	7
...	...
1023	1442

- ✓ Thus, only one outer page table and one inner page table are needed, which require 8KB!
- ✓ Note that two-level page table requires **three memory accesses** for each load/store (if TLB is not considered). So, there is a tradeoff between saving memory space and accelerating address translation speed.

□ More levels in hierarchical paging

➤ Two level paging is not appropriate for 64-bit machines

- ✓ Now a system has a 64-bit logical address space. If the page size is 4 KB (2^{12}), then a page table may consist of up to $2^{64}/2^{12}=2^{52}$ entries.
- ✓ If one level paging is used and each entry needs 4 bytes, then the page table with 2^{52} entries needs **16 PB**
- ✓ If two level paging is used, the outer page table still needs $4*2^{42}=16$ TB

outer page	inner page	offset
p_1	p_2	d
42	10	12

➤ What if three level paging?

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

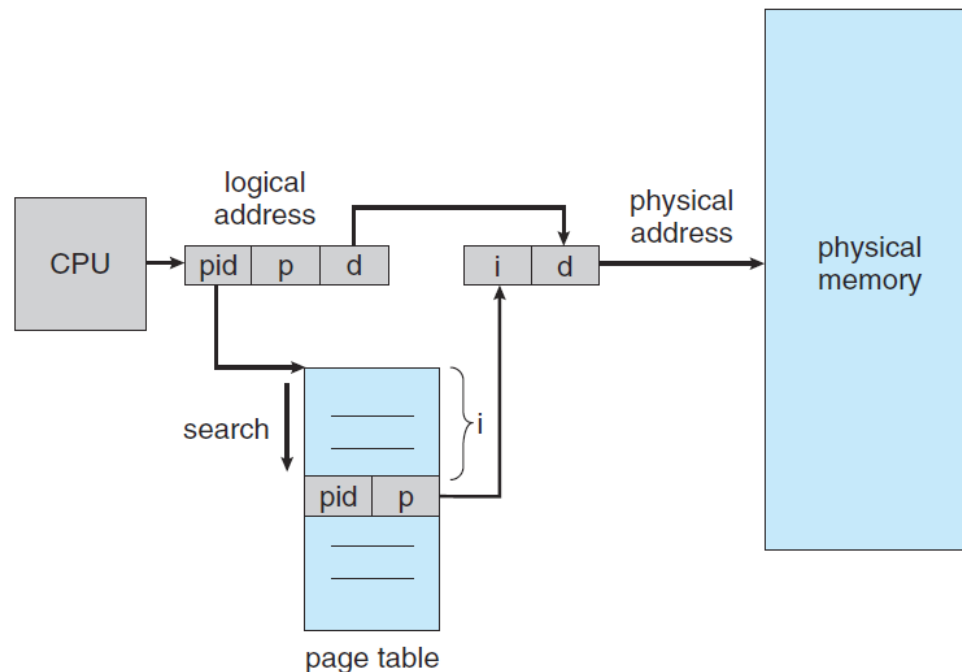
- ✓ The 2nd outer page table still needs $4*2^{32}=16$ GB

➤ We can go on increasing the number of paging levels to reduce the memory space for tables

- ✓ but each level requires a memory access
- ✓ n-level paging requires n+1 memory accesses for every load/store!
- ✓ need another page table implementation...

❑ Inverted Page Table

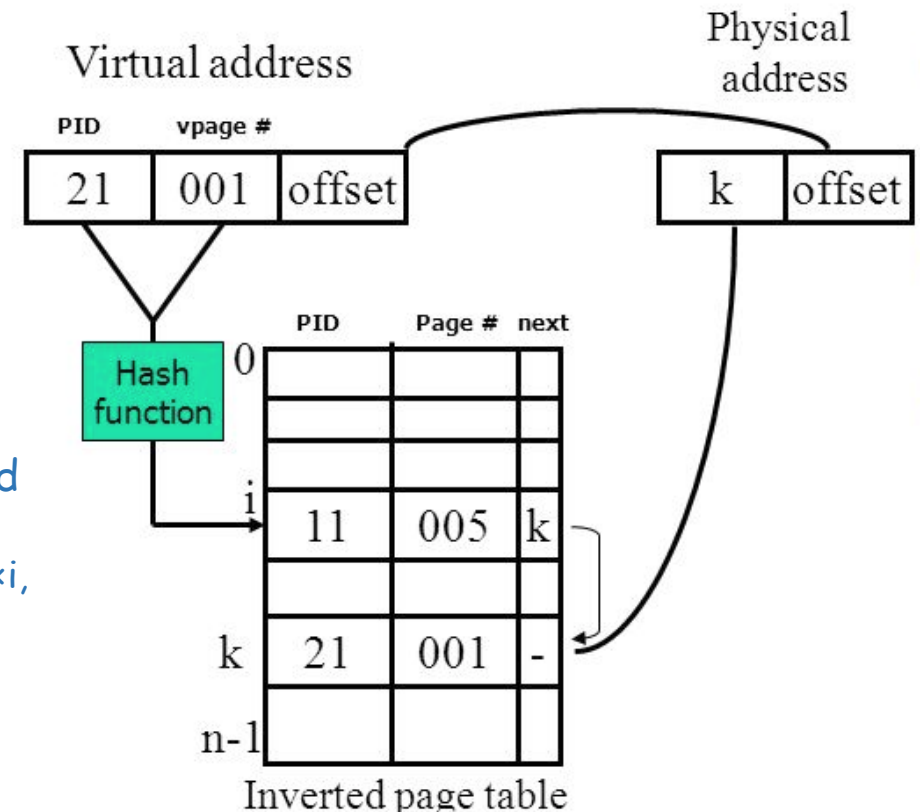
- An inverted page table has one entry for each frame/page of memory.
 - ✓ If the physical memory size is 4GB and the frame/page size is 4KB, the number of frames/pages = $4\text{GB}/4\text{KB} = 2^{20}$. That is, there are 2^{20} entries in the inverted page table. If each entry needs 4 bytes, then the inverted page table requires 4 MB memory space.
 - ✓ If the physical memory size is 8GB/16GB, the inverted page table requires 8MB/16MB memory space.
 - ✓ The size of inverted page table is determined by the physical memory size, NOT the total logical address space.



- Each **logical address** in the system consists of a triple **<process-id, page-number, offset>**.
- Each **inverted page-table entry** is a pair **<process-id, page-number>**
- For any address translation request, inverted page table is searched for a match **<pid, p>**.
- If a match is found—say, at entry **i**—then the physical address **<i, offset>** is generated.
- One inverted page table is shared among processes.
- Problem: long searching time.

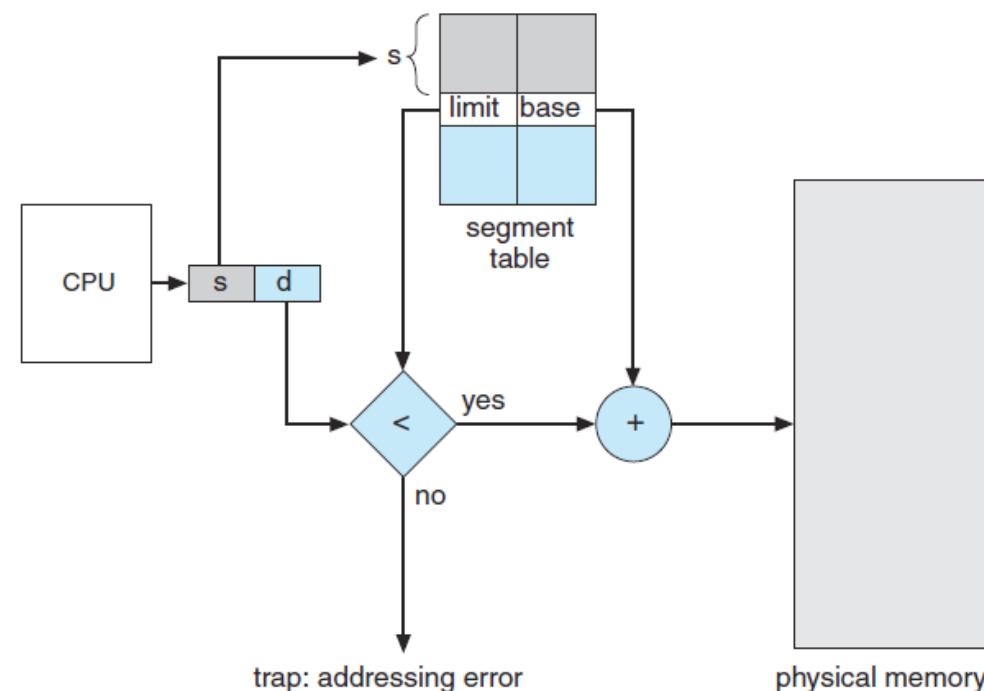
❑ Hashed inverted page table

- Basic idea: hash PID and page number, and the generated value is the index/address of the inverted page table page.
- Each entry in the new inverted page table comprises of a triple $\langle \text{PID}, \text{Page \#}, \text{NEXT} \rangle$
 - ✓ Next: pointer points to the next entry of the table.
 - ✓ The reason for adding "Next" is that different $\langle \text{PID}, \text{page\#} \rangle$ may hash into the same value.
- Address translation procedure
 1. Hash PID and page number, and find the related entry
 2. The PID and page number in the logical address are compared with the 1st field and 2nd field of the entry, respectively.
 - If a match is found—say, at entry i —then the physical address $\langle i, \text{offset} \rangle$ is generated.
 - Otherwise, go to the next entry (which is indicated by the 3rd field of the entry).



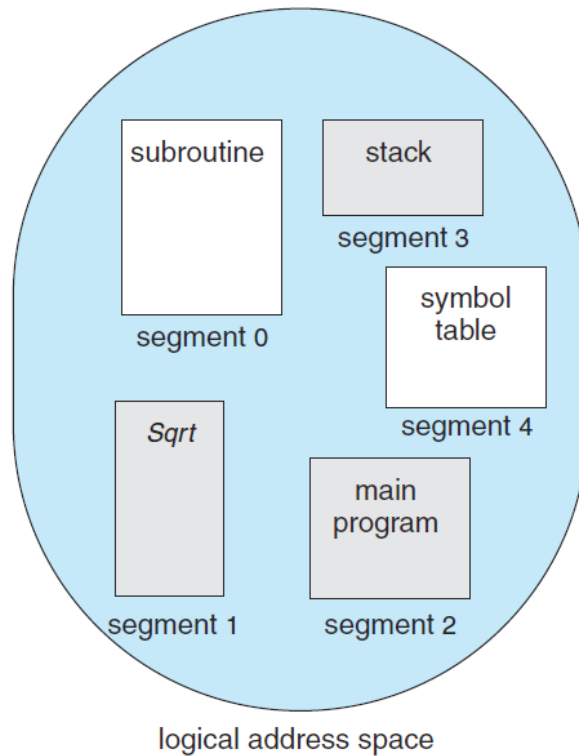
Segmentation

- ❑ Memory-management scheme that supports **user view of memory**
- ❑ A process is considered as a collection of **segments**
- ❑ Logical address is described as a pair: **<segment number (s), offset (d)>**
- ❑ Each process has a segment table (similar to the page table), which maps a logical address into a physical address.
 - Each entry in has a **segment base** and a **segment limit**
 - ✓ The **segment base** contains the starting physical address where the segment resides in memory
 - ✓ The **segment limit** specifies the length of the segment



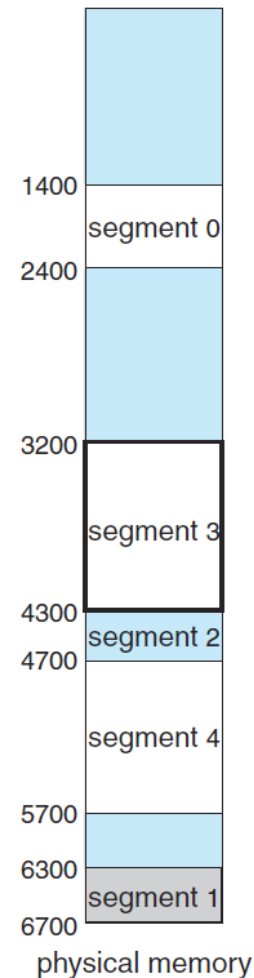
Segmentation

❑ Example of segmentation



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

segment table



❑ Comparison between paging and segmentation

- A page is of fixed block size, while a segment is of variable size.
- In paging, the user specified address is converted into a page number and offset. In segmentation, the user specifies each address by two quantities a segment number and the offset.
- In paging, the hardware decides the page size. In segmentation, the segment size is specified by the user.