Chapter 8: Memory Management

Presented by
NARZU TARNNUM
CSE, BU

What we will learn in Memory Management

- Background
- Swapping
- Memory Allocation
- Paging
- Segmentation

- Keep track of every memory location. Take decision which process will get how much memory and when.
- Update status of memory location: allocated or free.
- Swapping, Protection and security of memory space
- Address binding, Address translation
- Dynamic loading, linking.

Memory Management

- Main memory and registers built into the processor itself are the only storage that the CPU can access directly.
- So, any instructions in execution, and any data being used by the instructions, must be in one of these direct-access storage device.
- It is the functionality of an operating system which manages primary memory and moves processes back and forth between main memory and disk during execution.
- Goal is to ensure proper utilization of space and run long process using smaller area.
- Register access in one CPU clock (or less)
- Main memory can take many cycles, causing a **stall,** since it does not have the data required to complete the instruction that it is executing
- **Cache** sits between main memory and CPU registers for fast access
- Protection of memory required to ensure correct operation.

Protection of OS

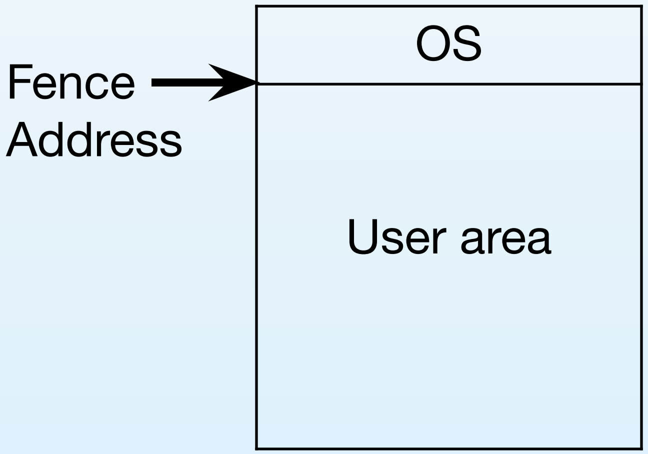
- The operating system has to be protected from access by user process.
- In, addition user process must be protected from one another.
- ~~This protection must be provided by the hardware.~~

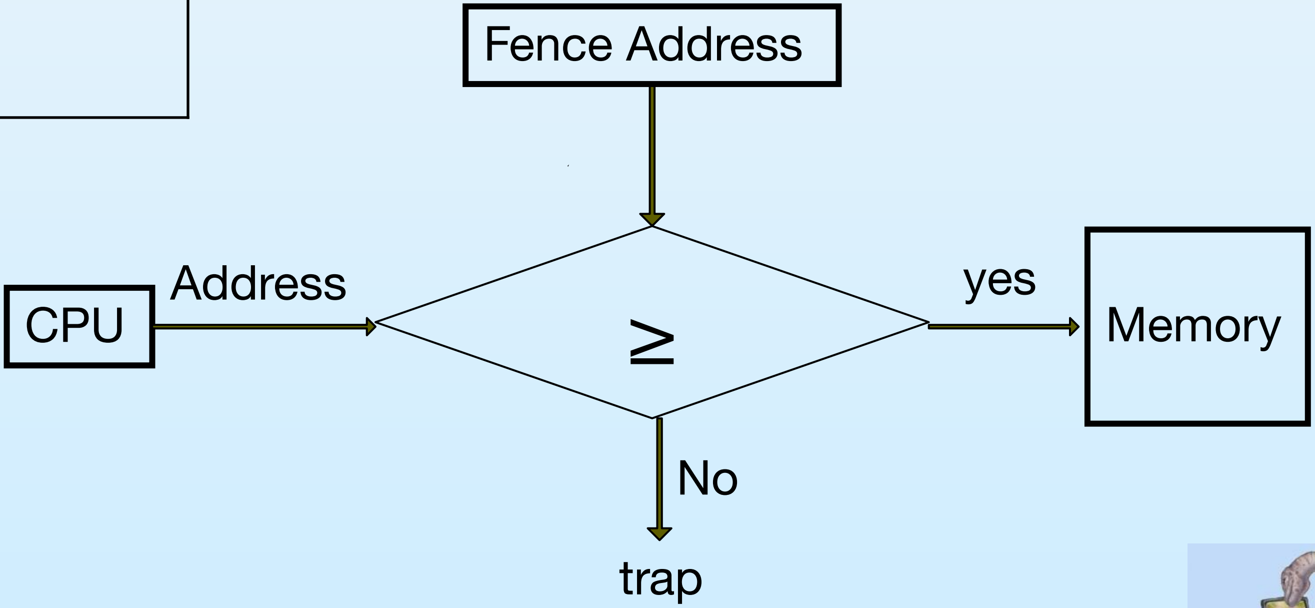
- To ensure that each process has a separate memory space.
- One process can't access OS and other process's memory location.

- In resident monitor memory model, there are two parts of memory:
  - Operating System area
  - User area

Hardware comparator for OS protection

- ✔ Fence Address
- ✔ Fence Register



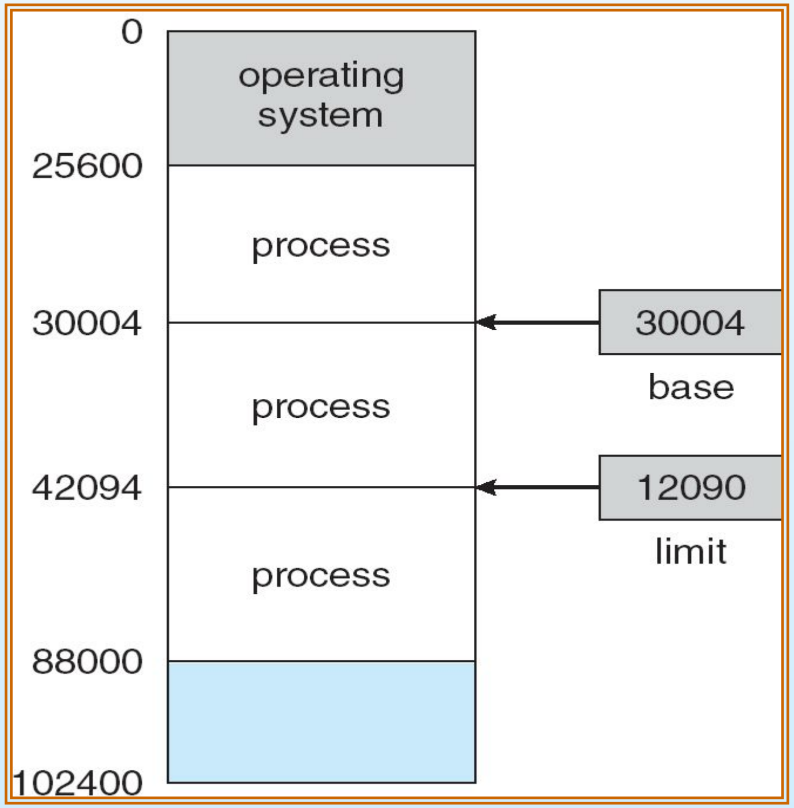


Basic Hardware

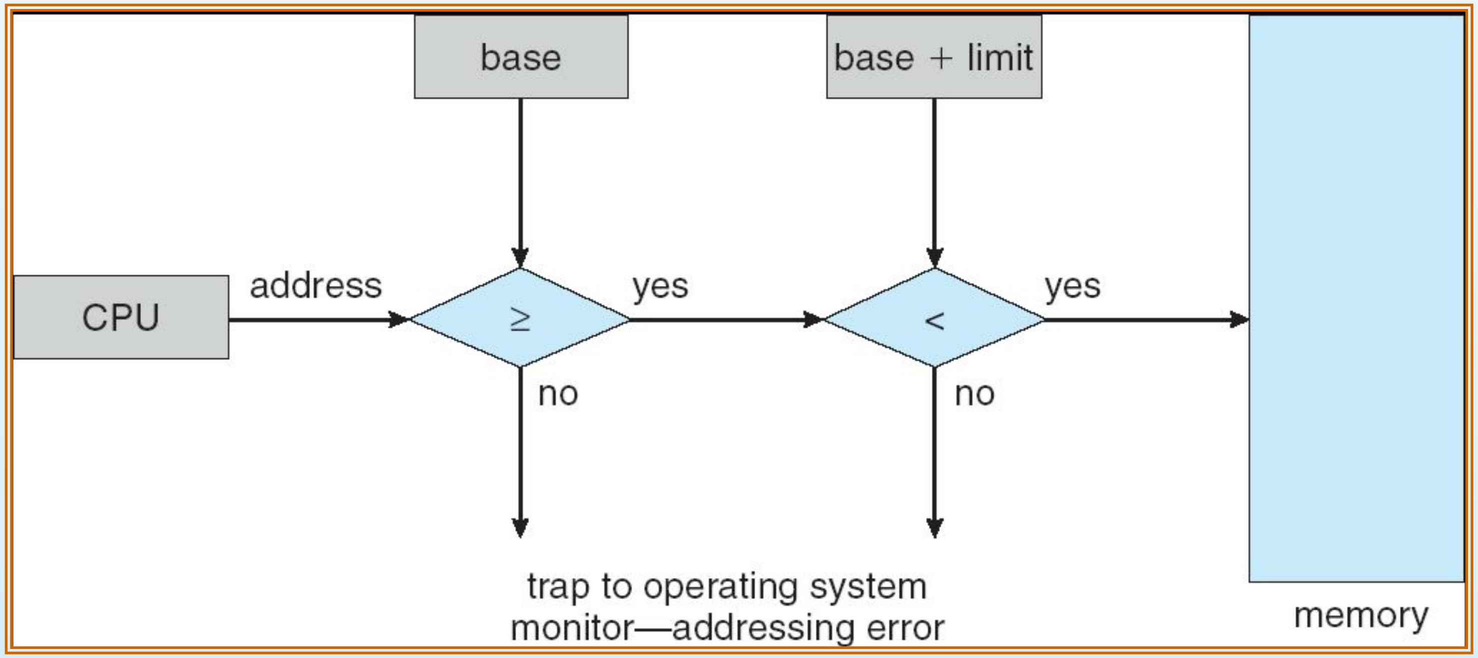
- Program must be brought into memory and placed within a process for it to be run.
- Make sure that each process has a separate memory space which ensure the legal address spaces. Two registers are used to provide protection:
    - **Base register** holds smallest legal physical memory address
    - **Limit register** holds size of the range
- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user

Base and a limit register

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



HW address protection with base and limit registers



Logical vs. Physical Address Space

- **Logical address** – an address generated by the CPU;
- **Physical address** – an address seen by the memory management unit.
- OR we can say: A logical address space is a set of logical addresses a computer generates for a specific program. A group of physical addresses mapped to corresponding logical addresses is called a physical address space.

❑ 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. In this case, we usually refer to the Logical address as a virtual address.

- **Logical address space** is the set of all logical addresses generated by a program.
- **Physical address space** is the set of all physical addresses corresponding to these logical addresses.

Memory Management Technique

- In operating system, following are four common memory management technique
  - Contiguous Allocation : the contiguous memory allocation assigns the consecutive blocks of memory to a process requesting for memory.
    - Fixed-partitioned –equal size allocation / Static/ MFT
    - Variable size partitioned/ dynamic/ MVT
  - It faces disadvantages-
  - Internal Fragmentation
  - External Fragmentation
  - Non-Contiguous Allocation : the noncontiguous memory allocation assigns the separate memory blocks at the different location in memory space in a nonconsecutive manner to a process requesting for memory.
    - Paged memory management
    - Segmented memory management

Fragmentation

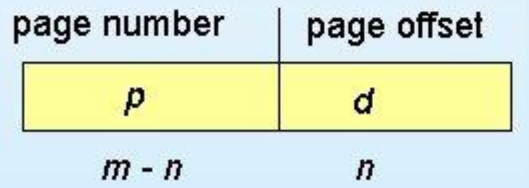
- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.

Paging

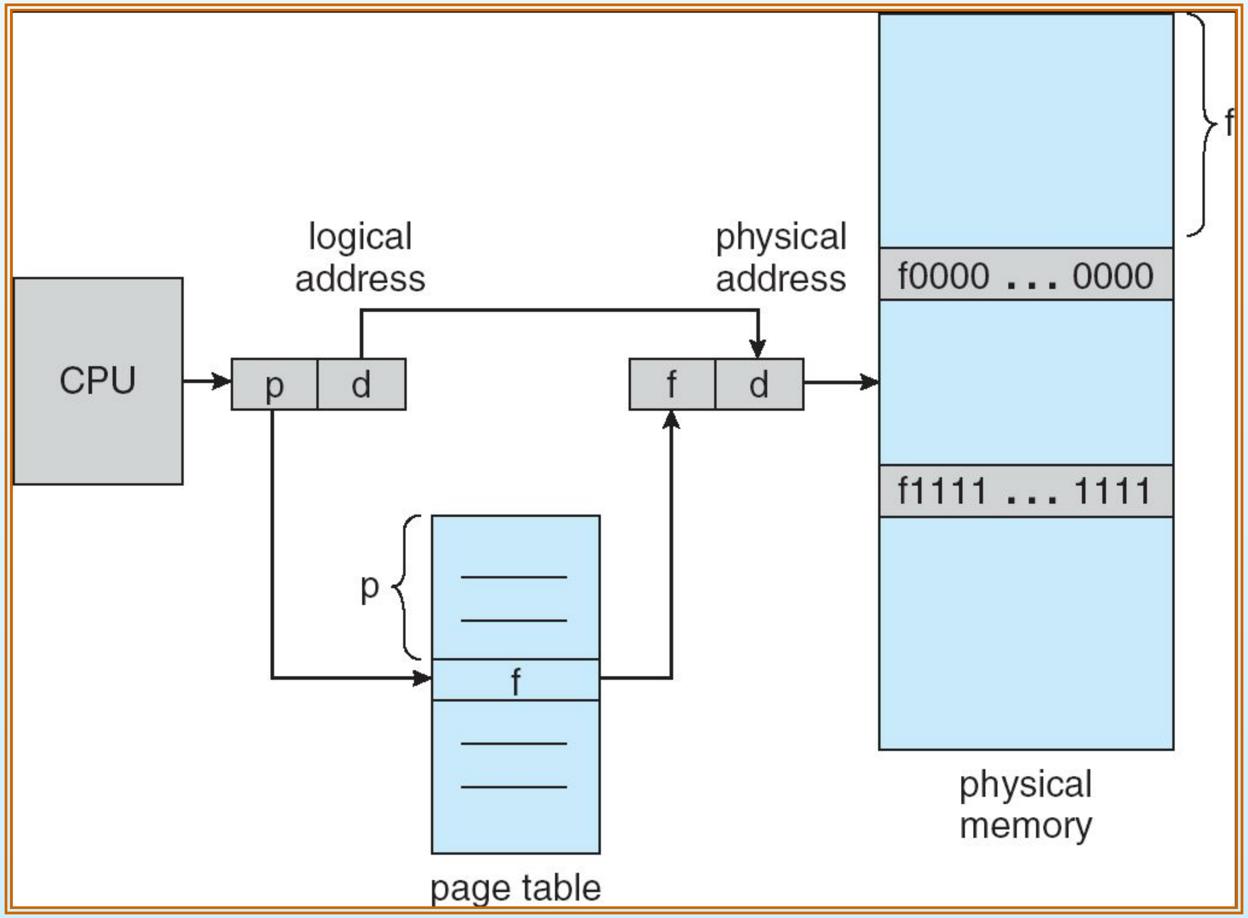
- Paging is another memory management scheme that offers a process to be non-contiguous. Paging avoids external fragmentation and the need for compaction.
- So, paging is a memory management scheme that permits the physical address space of a process to be non-contiguous.
- Divide physical memory into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 8192 bytes)
- Divide logical memory into blocks of same size called **pages**.
- Page size = Frame size
- To run a program of size *n* pages, need to find *n* free frames and load program
- When a process is to be execute, its pages are loaded into any available memory frames from the backend store.
- Set up a page table to translate logical to physical addresses
- **Internal fragmentation**

Address Translation Scheme

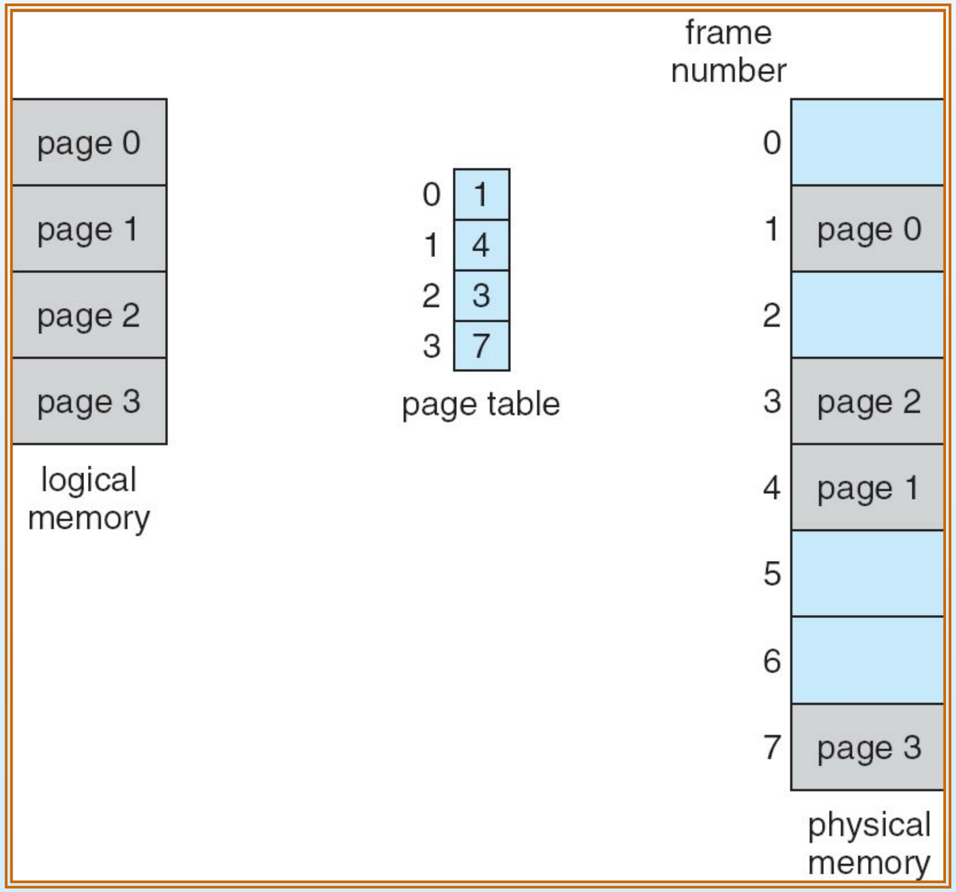
- Translation of Logical Address to Physical Address using page table.
- Address generated by CPU is divided into two parts:
  - *Page number (p)* – used as an index into a *page table* which contains base address of each page in physical memory
  - *Page offset (d)* – is the displacement within the page. combined with base address to define the physical memory address.



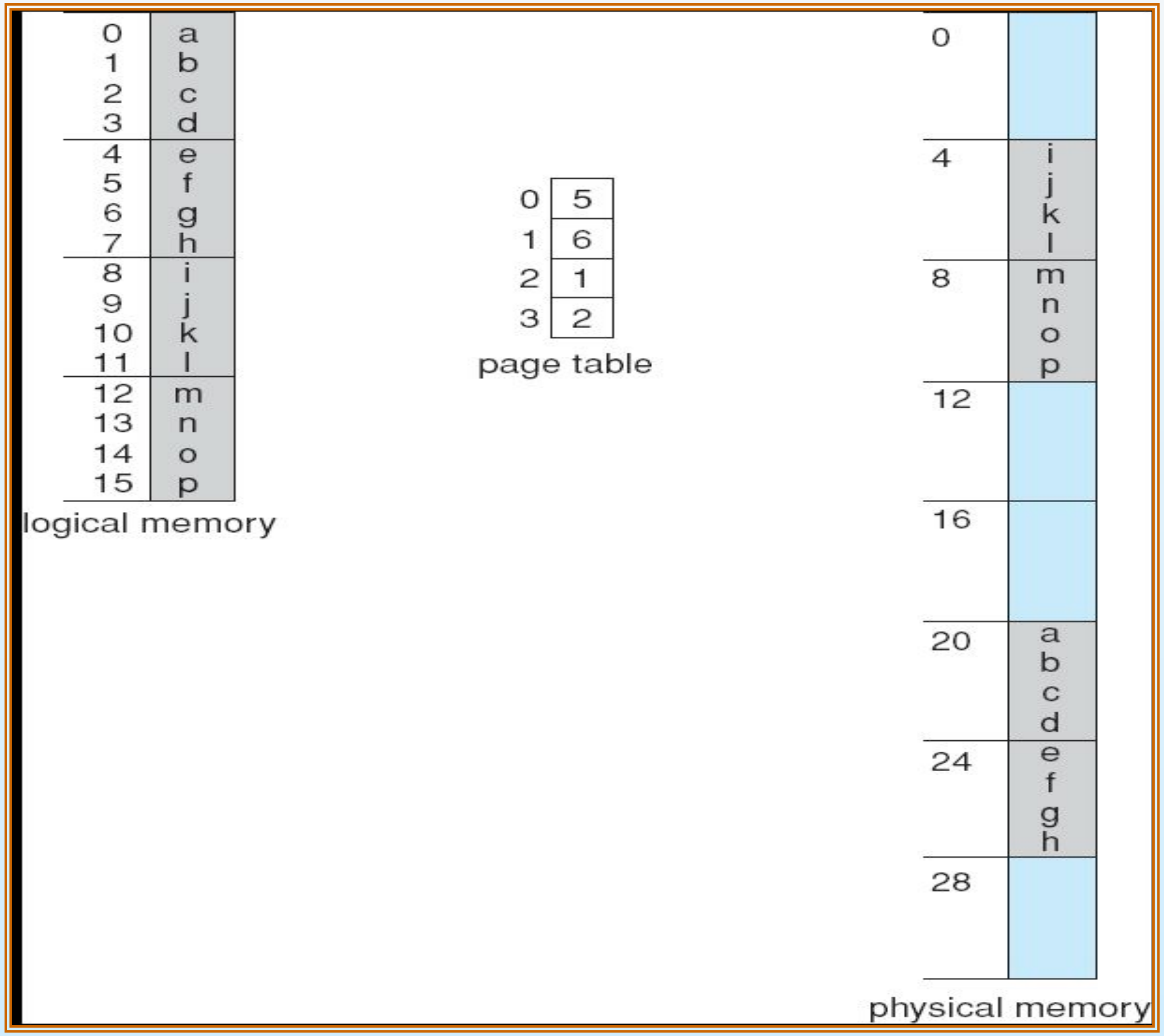
Address Translation Architecture/ Paging h/w



Paging Example



Paging Example



Paging Example from book

As a concrete (although minuscule) example, consider the memory in Figure 8.12. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages), we show how the programmer's view of memory can be mapped into physical memory. Logical address 0 is page 0, offset 0. Indexing into the page table, we find that page 0

is in frame 5. Thus, logical address 0 maps to physical address 20 [= (5 × 4) + 0]. Logical address 3 (page 0, offset 3) maps to physical address 23 [= (5 × 4) + 3]. Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6. Thus, logical address 4 maps to physical address 24 [= (6 × 4) + 0]. Logical address 13 maps to physical address 9.

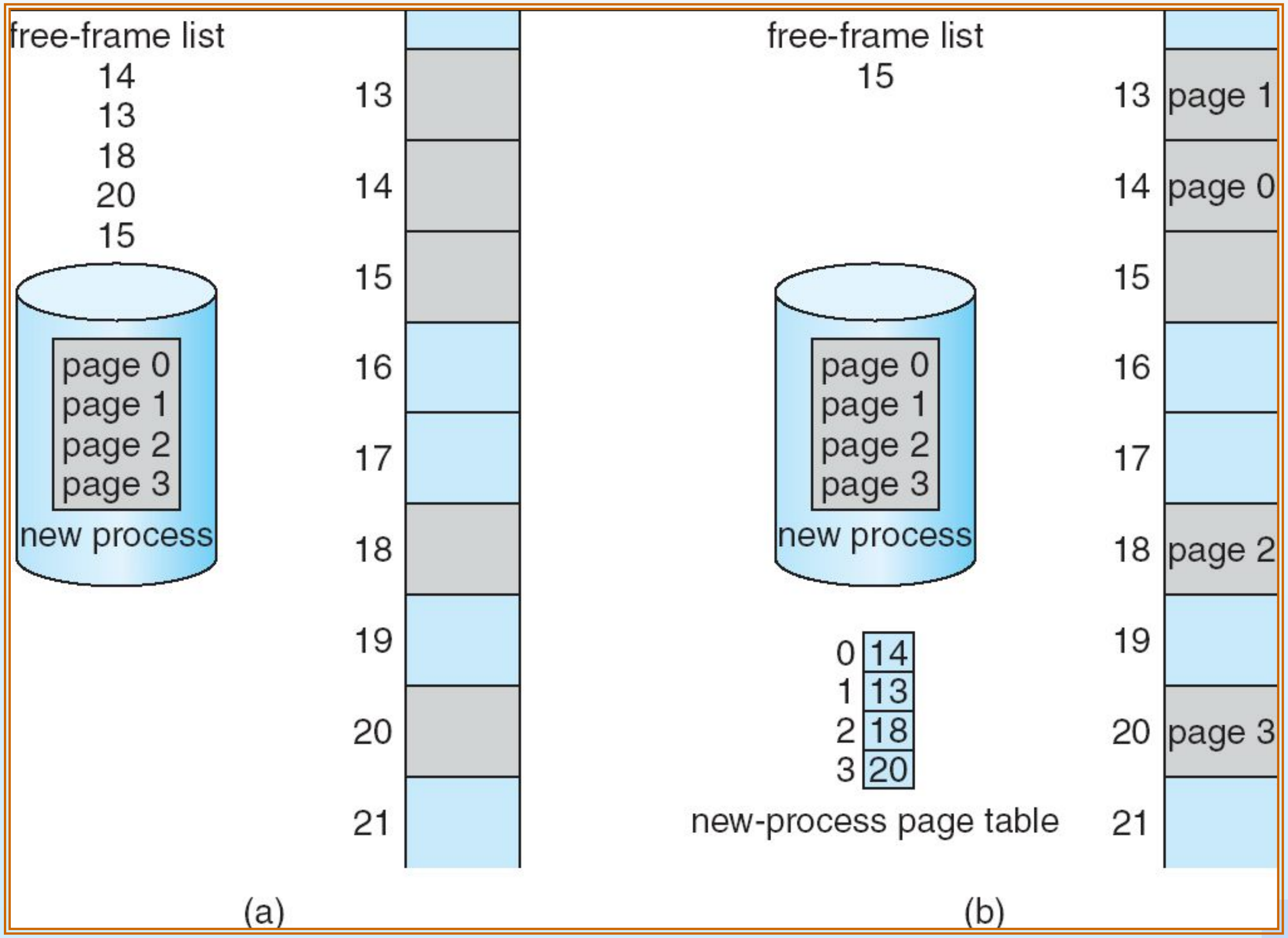
Discussing limitations of paging

When we use a paging scheme, we have no external fragmentation: any free frame can be allocated to a process that needs it. However, we may have some internal fragmentation. Notice that frames are allocated as units. If the memory requirements of a process do not happen to coincide with page boundaries, the last frame allocated may not be completely full. For example, if page size is 2,048 bytes, a process of 72,766 bytes will need 35 pages plus 1,086 bytes. It will be allocated 36 frames, resulting in internal fragmentation of 2,048 − 1,086 = 962 bytes. In the worst case, a process would need n pages plus 1 byte. It would be allocated n + 1 frames, resulting in internal fragmentation of almost an entire frame. If process size is independent of page size, we expect internal fragmentation to average one-half page per process

Paging (Cont.)

- Calculating internal fragmentation
  - Page size = 2,048 bytes
  - Process size = 72,766 bytes
  - 35 pages + 1,086 bytes= 36 frames needed
  - 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 sizes desirable?
  - But each page table entry takes memory to track
  - Page sizes growing over time
    - Solaris supports two page sizes – 8 KB and 4 MB
- Process view and physical memory now very different
- By implementation process can only access its own memory

Free Frames



free-frame list
14
13
18
20
15
page 0
page 1
page 2
page 3
new process
13
14
15
16
17
18
19
20
21
(a)
free-frame list
15
page 0
page 1
page 2
page 3
new process
0 14
1 13
2 18
3 20
new-process page table
13 page 1
14 page 0
15
16
17
18 page 2
19
20 page 3
21
(b)

Hardware Implementation of Page Table

1st case:

- We can implement the page table as a set of dedicated registers.
- But this can be used only when page table is reasonably small.

Hardware Implementation of Page Table

2nd case:

- Page table is kept in main memory and
- **Page-table base register** (**PTBR**) points to the page table
- **Page-table length register** (**PTLR**) indicates size of the page table
- In this scheme every data/instruction access requires two memory accesses
  - One for the page table and one for the data / instruction

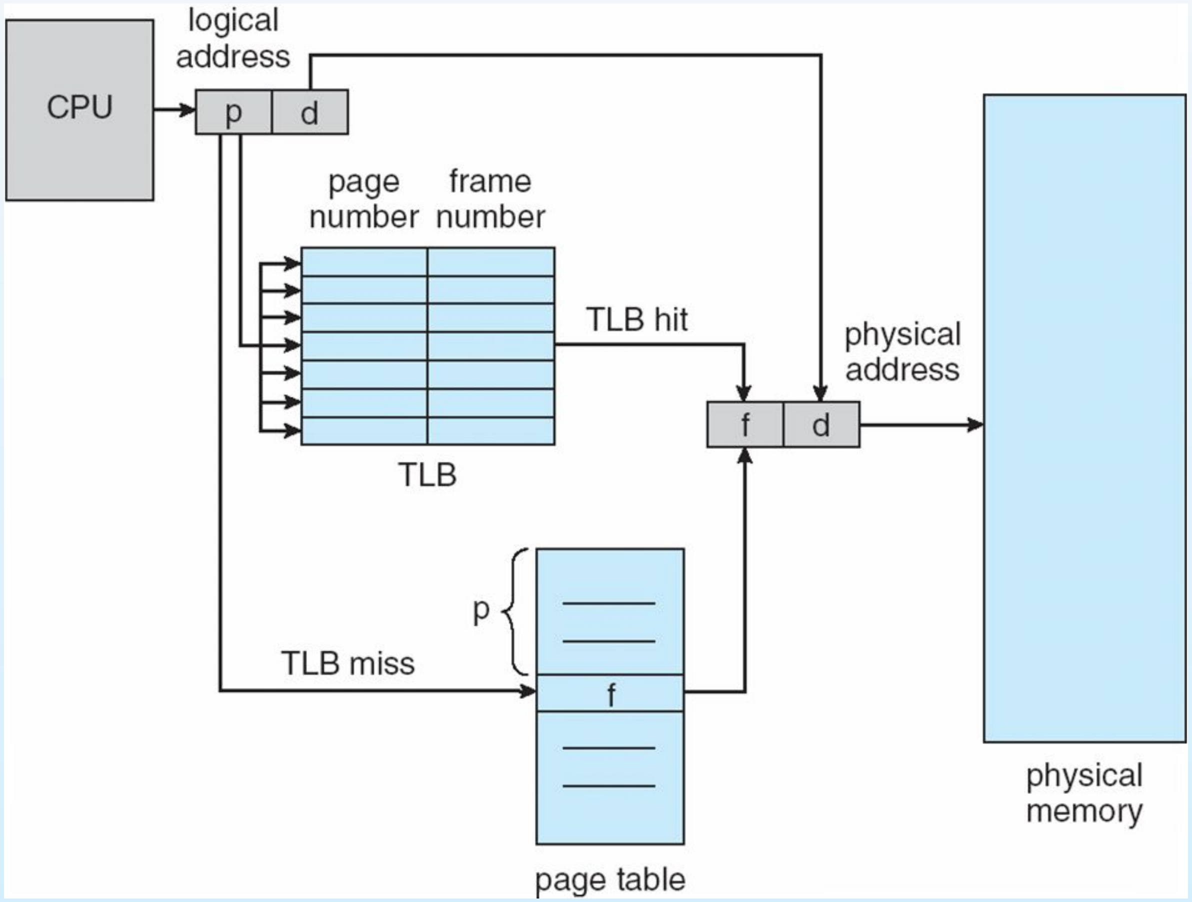
----

- **The two memory access problem can be solved** by the use of a special fast-lookup hardware cache called **associative or high speed memory** or **translation look-aside buffers** (**TLBs**)
- It consists of two parts: A key, A value.
- **The search is fast.**

Hardware Implementation of Page Table

- TLB contents only a few of the page table entries.
- When a logical address is generated by the CPU, its page number is presented to the TLB.
- If the page number is found, its frame number is immediately available and used to access memory □TLB Hit
- If the page number is not in the TLB, a memory reference to the page table must be made □TLB Miss
- When the frame number is obtained, we can use it to access memory.
- Also we add the page number and frame number to the TLB on a TLB miss, so that they will be found quickly on the next reference.
- TLBs typically small (64 to 1,024 entries)
- Replacement policies must be considered
- Some entries can be **wired down** for permanent fast access

Paging Hardware With TLB



Effective Memory Access Time

- The percentage of times that the page number of interest is found in the TLB is called the hit ratio. An 80-percent hit ratio, for example, means that we find the desired page number in the TLB 80 percent of the time. If it takes 100 nanoseconds to access memory, then a mapped-memory access takes 100 nanoseconds when the page number is in the TLB. If we fail to find the page number in the TLB then we must first access memory for the page table and frame number (100 nanoseconds) and then access the desired byte in memory (100 nanoseconds), for a total of 200 nanoseconds. (We are assuming that a page-table lookup takes only one memory access, but it can take more, as we shall see.)

Effective Memory Access Time

- associative Lookup = ε time unit
  - Can be < 10% of memory access time
- Hit ratio = α
  - Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
- Consider α = 80%, ε = 20ns for TLB search, 100ns for memory access
  - EAT = 0.80 x 120 + 0.20 x 220 = 140ns
- Consider more realistic hit ratio -> α = 99%, ε = 20ns for TLB search, 100ns for memory access
  - EAT = 0.99 x 120 + 0.01 x 220 = 121ns

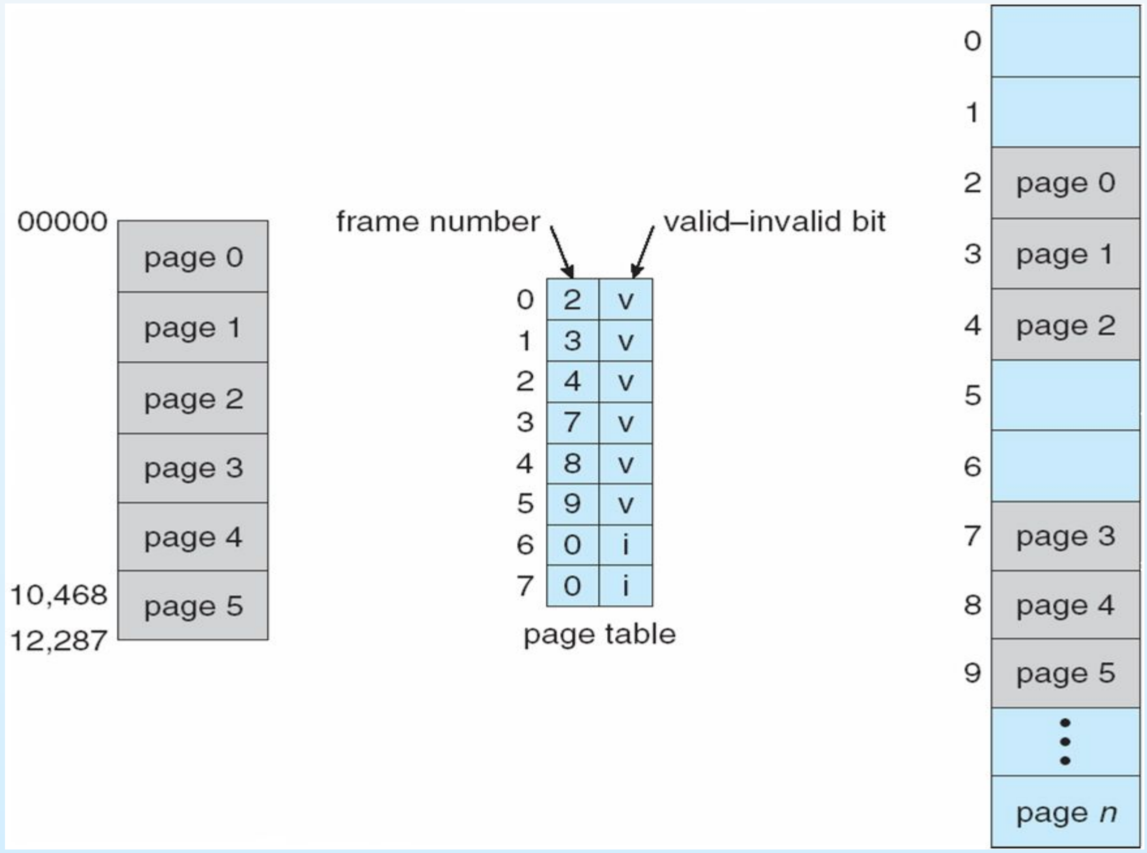
$$EAT\ of\ MM = TLB\ hit + TLB\ miss$$

$$= \{\propto X\ (ma\ time + \in)\} + \{(1 - \propto) X\ (2ma\ time + \in)\}$$

Memory Protection

- Memory protection implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed
  - Can also add more bits to indicate page execute-only, and so on
- **Valid-invalid** bit attached to each entry in the page table:
  - "valid" indicates that the associated page is in the process' logical address space, and is thus a legal page
  - "invalid" indicates that the page is not in the process' logical address space
  - Or use **page-table length register** (**PTLR**)
- Any violations result in a trap to the kernel

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



Shared Pages

- An advantage of paging is the possibility of sharing common code.
- This is particularly important in a time sharing environment.

- Consider a system that supports 40 users, each of whom executes a text editor.
- If the text editor consists of 150KB of code and 50KB of data space, we need:

  (150 X 40) + (50 X 40)

  = 6000 + 2000

  = 800KB to support the 40 users

Shared Pages

- However, if the code is reentrant code (or pure code), it can be shared among different processes
  - Reentrant code means the code that is non-self-modifying code. It never changes during executions.
- Thus two or more processes can execute the same code at the same time.
- Each process has its own copy of registers and data storage to hold the data for the process's execution.
- The data for two different processes will, of course, be different.

Shared Pages

- An advantage of paging is the possibility of sharing common code.
- This is particularly important in a time sharing environment.

- Consider a system that supports 40 users, each of whom executes a text editor.
- If the text editor consists of 150KB of code and 50KB of data space, we need:

  (150 X 40) + (50 X 40)

  = 6000 + 2000

  = 8000KB to support the 40 users

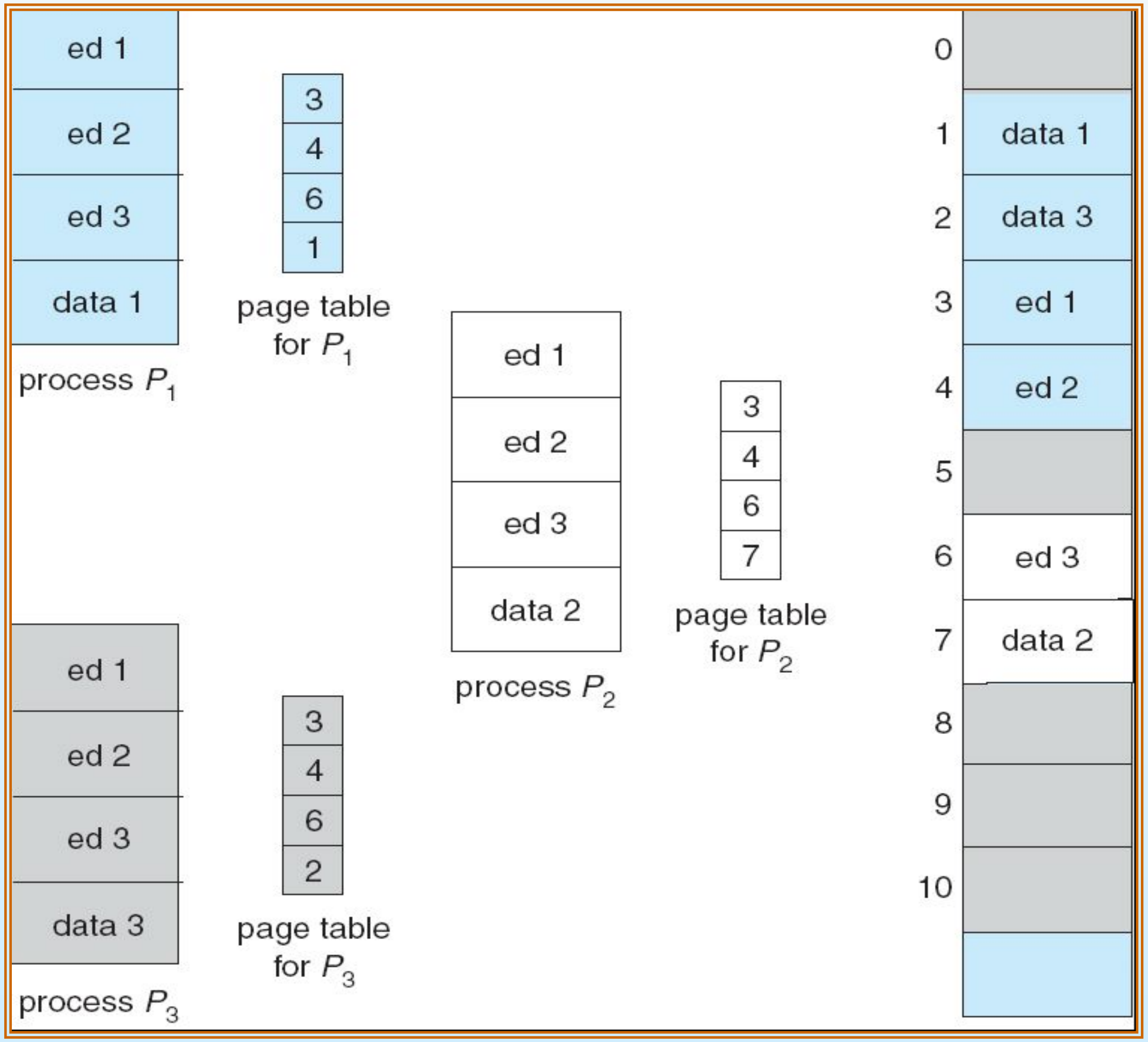
Now, if the code is shared, we need:

150 + (50 X 40)

= 150 + 2000

= 2150 to support 40 users

Shared Pages Example



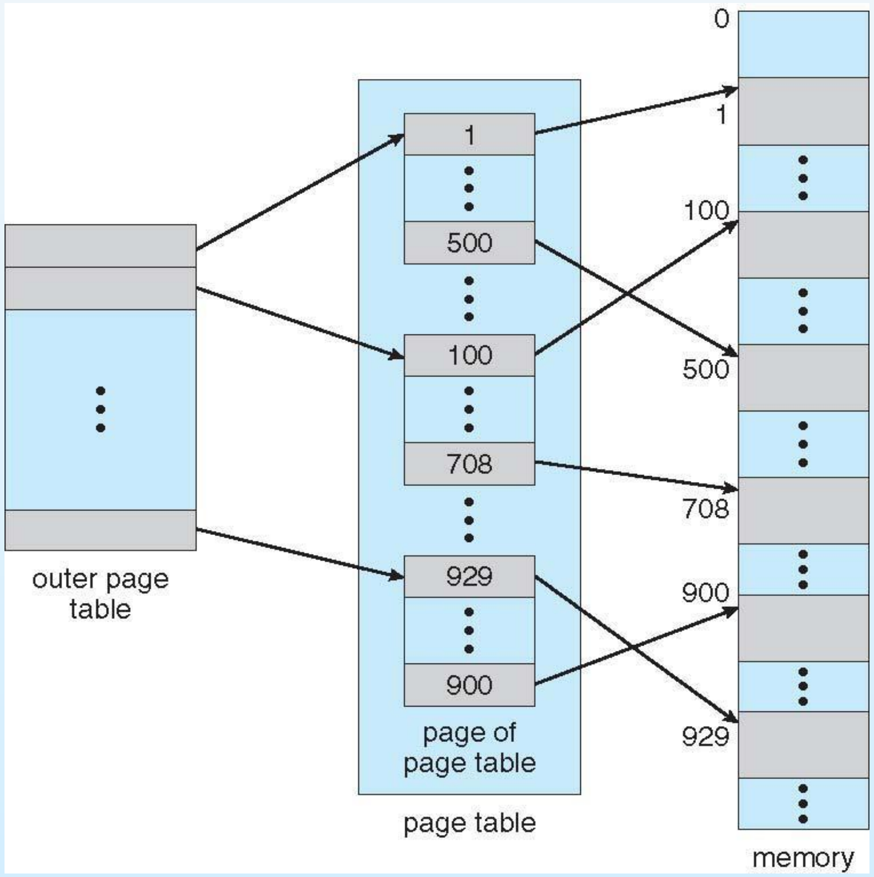
Operating Systems
Hierarchical Page Table

Structure of the Page Table

- Memory structures for paging can get huge using straightforward methods
  - Consider a 32-bit logical address space as on modern computers
  - Page size of 4 KB (2^{12})
  - Page table would have 1 million entries (2^{32} / 2^{12})
  - If each entry is 4 bytes -> 4 MB of physical address space / memory for page table alone
    - That amount of memory used to cost a lot
    - Don't want to allocate that contiguously in main memory

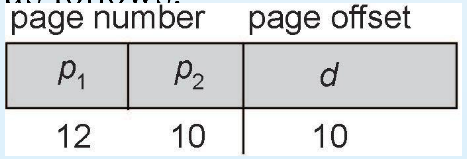
Hierarchical Page Tables

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



Two-Level 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_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table
- Known as **forward-mapped page table**

End of Chapter 8