操作系统原理与设计

第 8 章 Main Memory2

陈香兰

中国科学技术大学计算机学院

2009 年 09 月 01 日

提纲

Discrete Memory Allocation

- paging（分页）
  - internal fragmentation <one page
- segmentation（分段）
  - logical
- combined paging & segmentation（段页式）

Outline

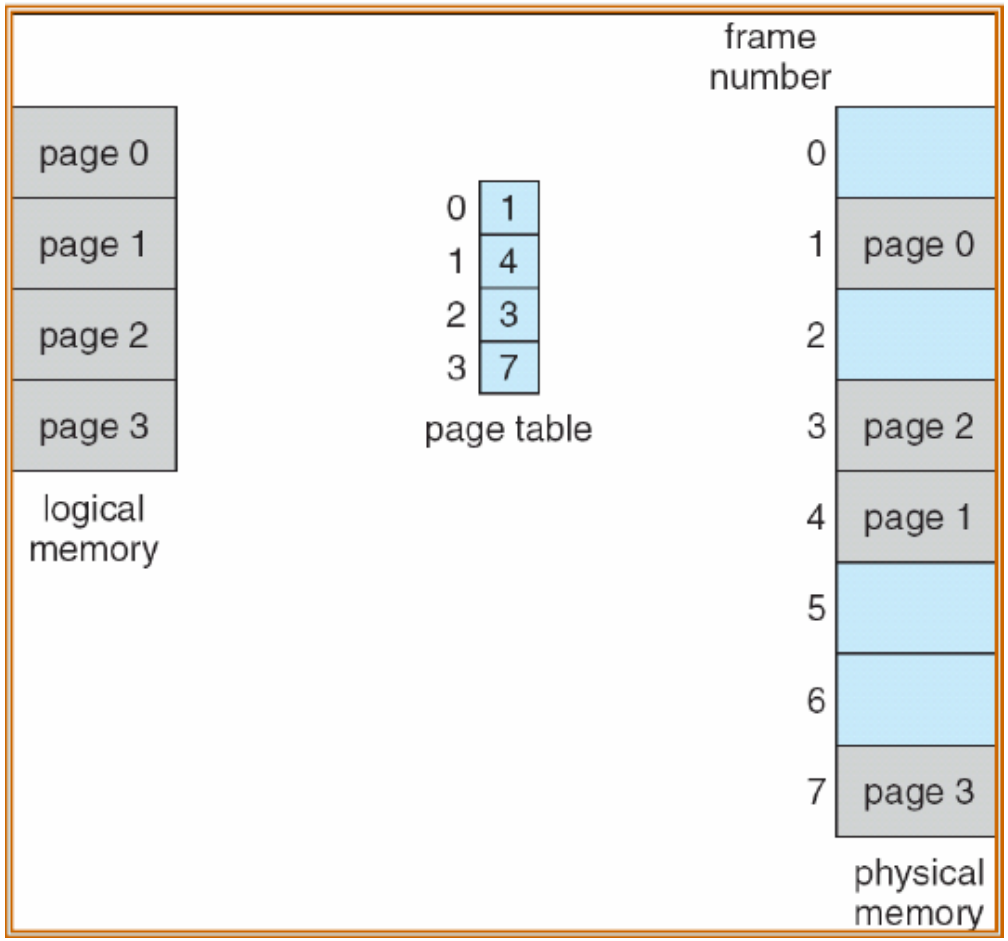
Paging I

- Logical address space of a process can be **noncontiguous**; process is allocated physical memory whenever the latter is available
- **Basic Method**
  1. Divide **physical memory** into fixed-sized blocks called **frames**
     - size is power of 2, between 512B and 8,192B
     - Page Frame Number, **PFN**: $0, 1, ..., PFN_{max}$
  2. Divide **logical memory** into blocks of **same size** called **pages**
     - Logical Frame Number, **LFN**: $0, 1, ..., LFN_{max}$
  3. the **backing store** is also divided into fixed-sized blocks of same size as frames

Paging II

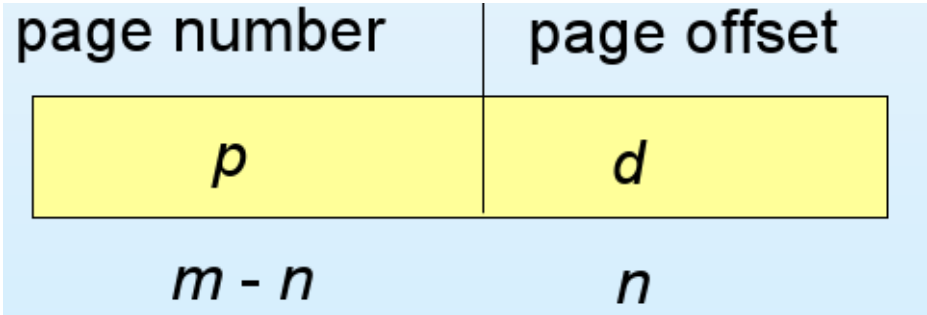
- need hardware and software support for paging
  - **Keep track** of all free frames
  - To run a program of size *n* pages, need to **find** *n* free frames and **load** program
  - Set up a **page table** to translate logical to physical addresses for **each process**
- **Internal fragmentation** $<$ page size

Paging Model of Logical and Physical Memory



Address Translation Scheme

- Address generated by CPU is divided into:
  - **Page number (p), LFN**
  - **Page offset (d)**



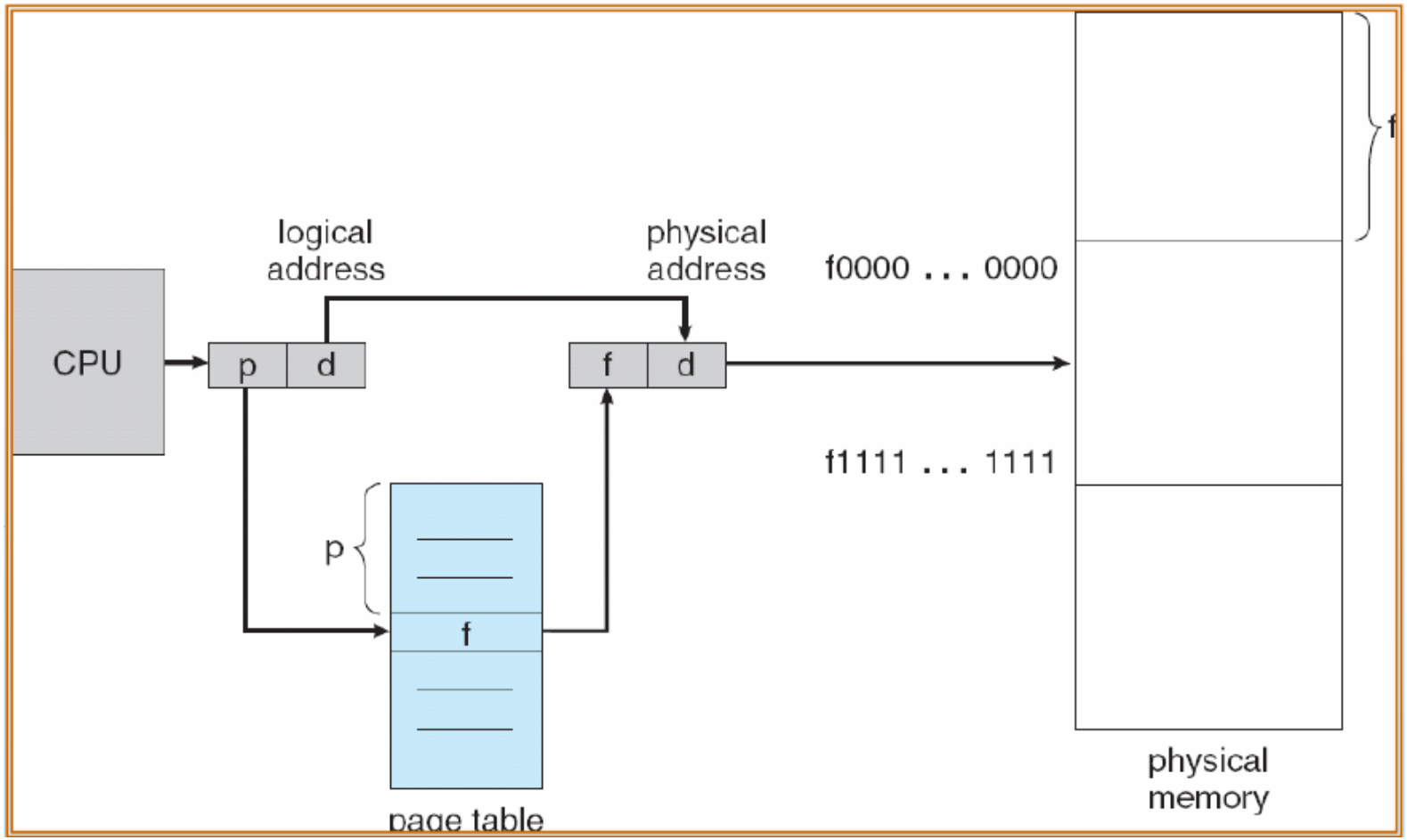
For given logical address space 2^m and page size 2^n

- for 32bits system & 4KB page size, $m = 32$, $n = 12$, $m - n = 20$

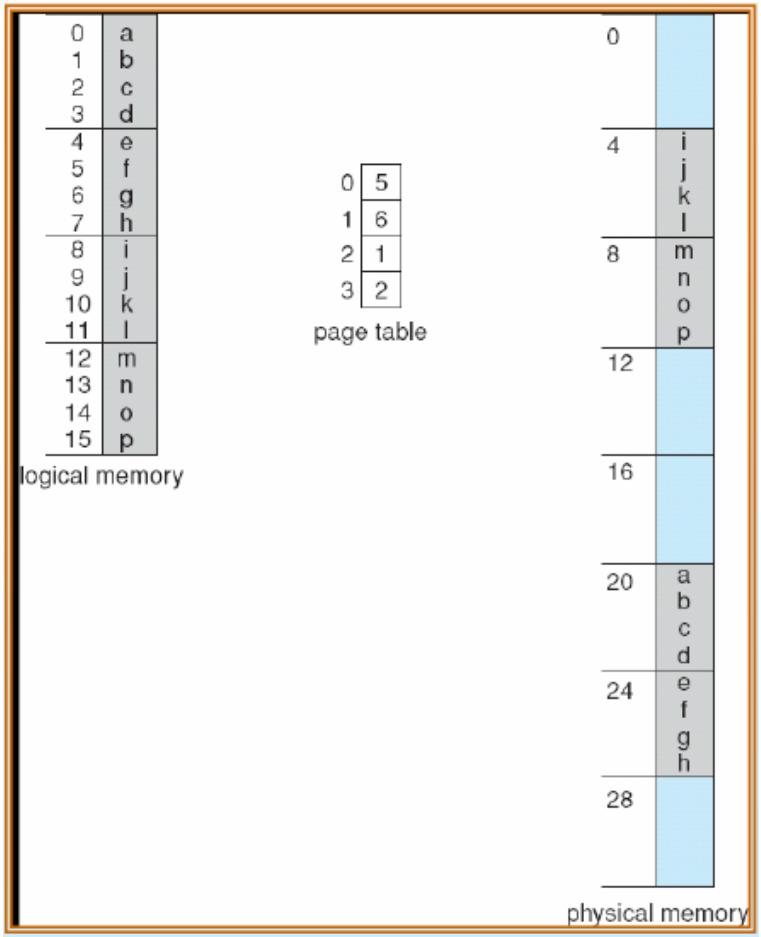
- 计算页（页框）号和页（页框）内偏移的方法：
  - 地址：A
  - 页（页框）大小：L
    - 页（页框）号$p = A$ 整除 L
    - 页（页框）内偏移$d = A \ mod \ L$
  - 考虑 L 是 2 的幂，不妨设为 2^N，则
    - p=A右移N位，即取 A 的高 $(32-N)$ 位
    - d=A的低端N位

Paging Hardware

- $LFN\,(p) + offset\,(d) \longrightarrow PFN\,(f) + offset\,(d)$



Paging Example

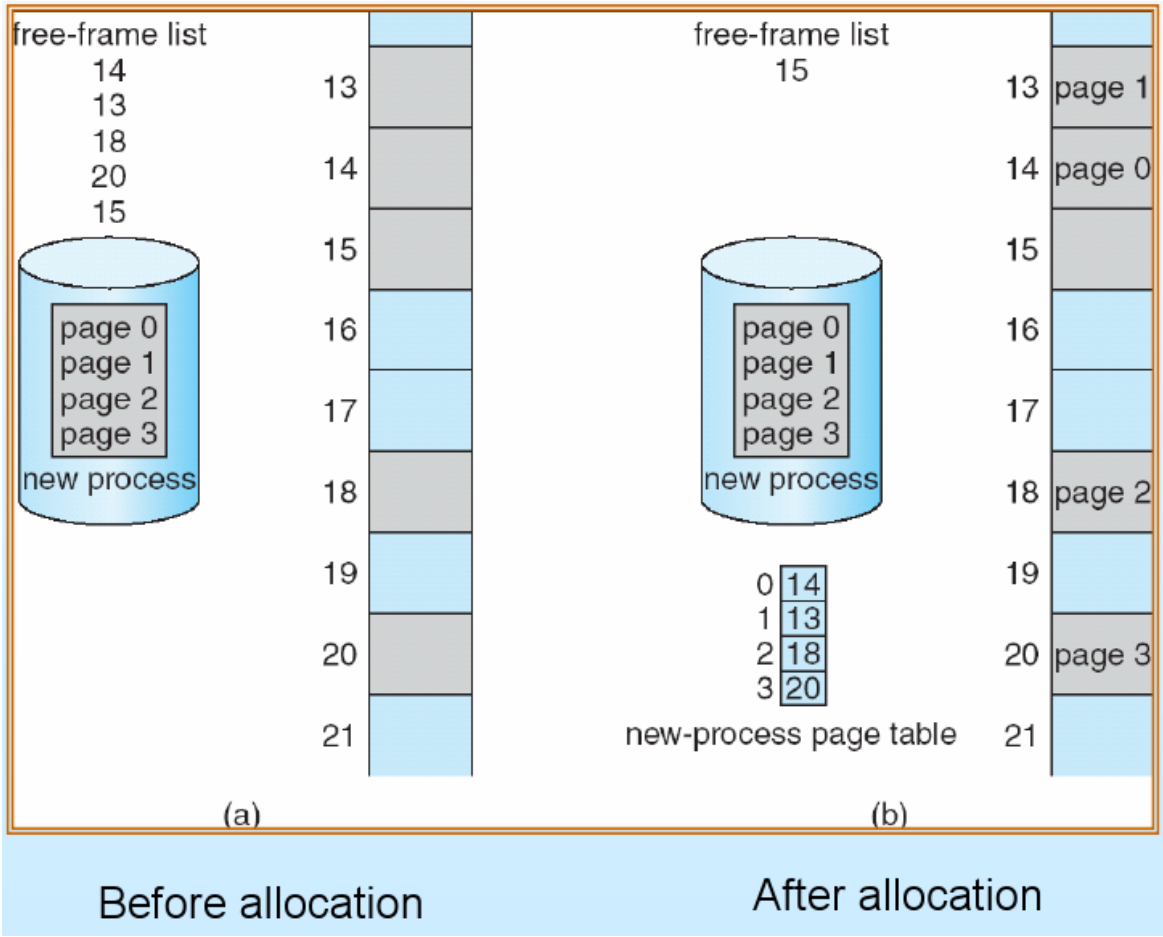


32-byte memory and 4-byte pages

Free Frames I

- Since OS is managing physical memory, it must be aware of the allocation details of physical memory
  - which frames are allocated
  - which frames are available
  - how many total frames
  - ...
- frame table: one entry for each physical page frame

Free Frames II

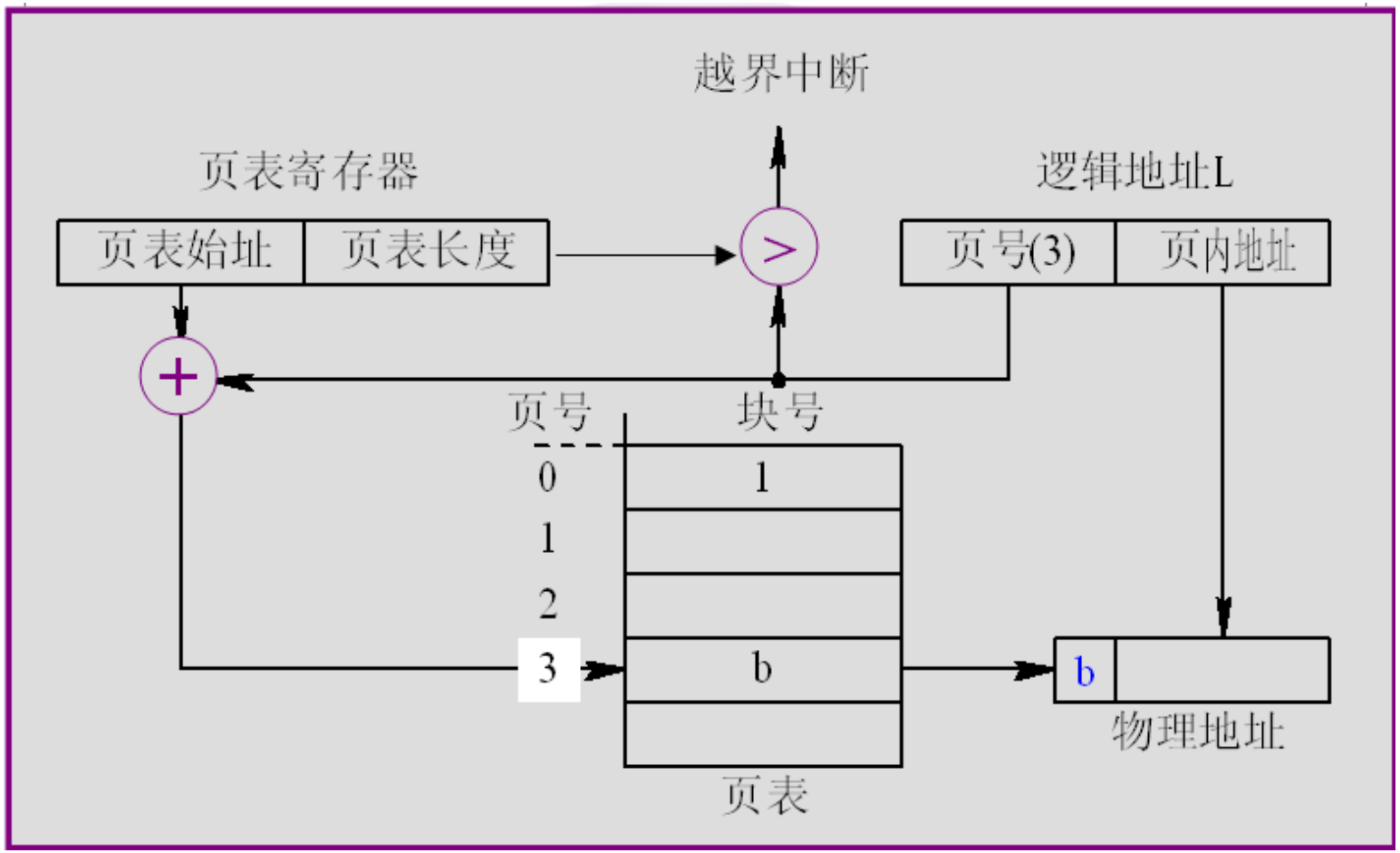


Outline

Hardware support I

- special hardware (software) is needed to implement page table
- Hardware support
  - basic paging hardware
  - paging hardware with TLB
- Implementation of Page Table : **basic paging hardware**
  - 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
  - context switch ?

Hardware support II



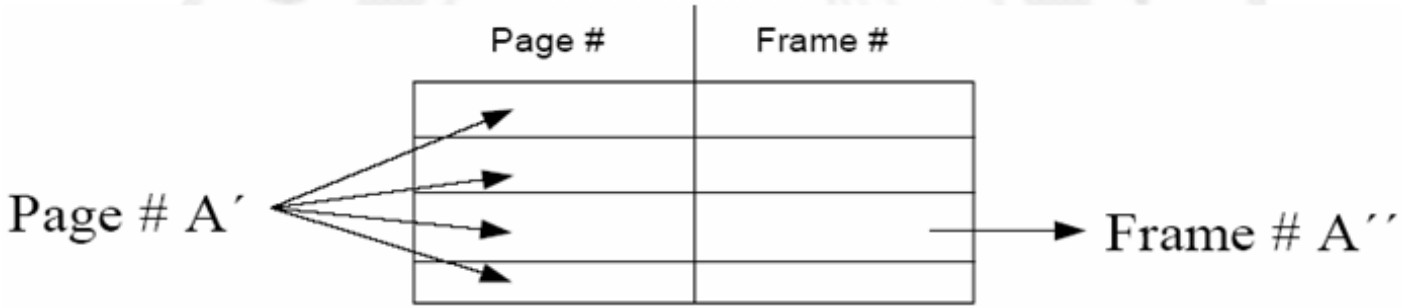
Hardware support III

- Effective memory-access time
  - In this scheme every data/instruction access requires **two memory accesses**.
  - One for the **page table** &
  - one for the **data/instruction**.

- Solution to two memory access problem:
  - a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers (TLBs)**

Hardware support IV

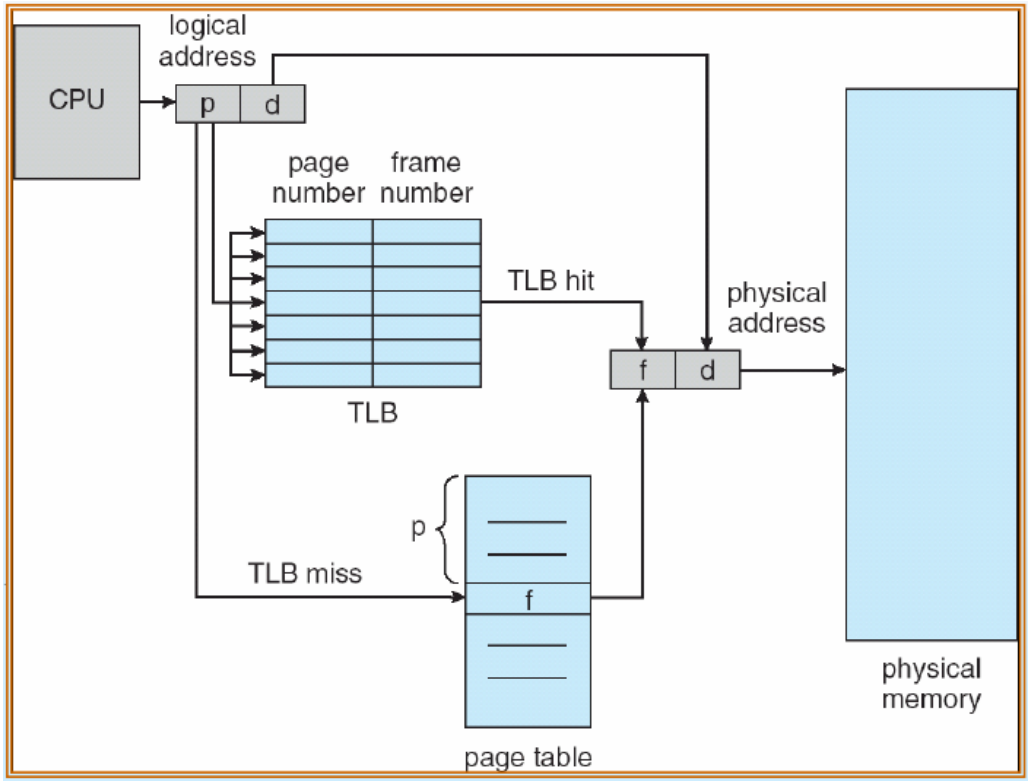
- **Associative Memory**
  - Each register: a key & a value
  - **parallel search** (high speed)
  - Expensive, typically 8~2048 entries



Address translation (A', A")

- If A' is in associative register, get frame # out
- Otherwise get frame # from page table in memory

Paging Hardware With TLB



- Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry
  - uniquely identifies each process to provide address-space protection for that process

- **TLB miss（TLB 缺失）**
  - If the page number is not in the associative registers
    - Get & store
- **Hit ratio（命中率）**
  - The percentage of times that a page number is found in the associative registers
  - ratio related to number of associative registers
- Context switch→**TLB flushed**
- **TLB replacement** algorithm

Effective Access Time

- if
  - Associative Lookup $= \epsilon$ time unit
  - Assume memory cycle time is t microsecond
  - Hit ratio $= \alpha$
- then **Effective Access Time** (EAT)

$$\begin{aligned} EAT &= (t+\epsilon)\,\alpha + (2t+\epsilon)\,(1-\alpha) \\ &= 2t + \epsilon - t\alpha \end{aligned}$$

- If $\epsilon = 20ns,\ t = 100ns,\ \alpha 1 = 80\%,\ \alpha 2 = 98\%$:
  - if TLB hit: $20 + 100 = 120ns$
  - if TLB miss: $20 + 100 + 100 = 220ns$
  - $EAT1 = 120 * 0.8 + 220 * 0.2 = 140ns$
  - $EAT2 = 120 * 0.98 + 220 * 0.02 = 122ns$

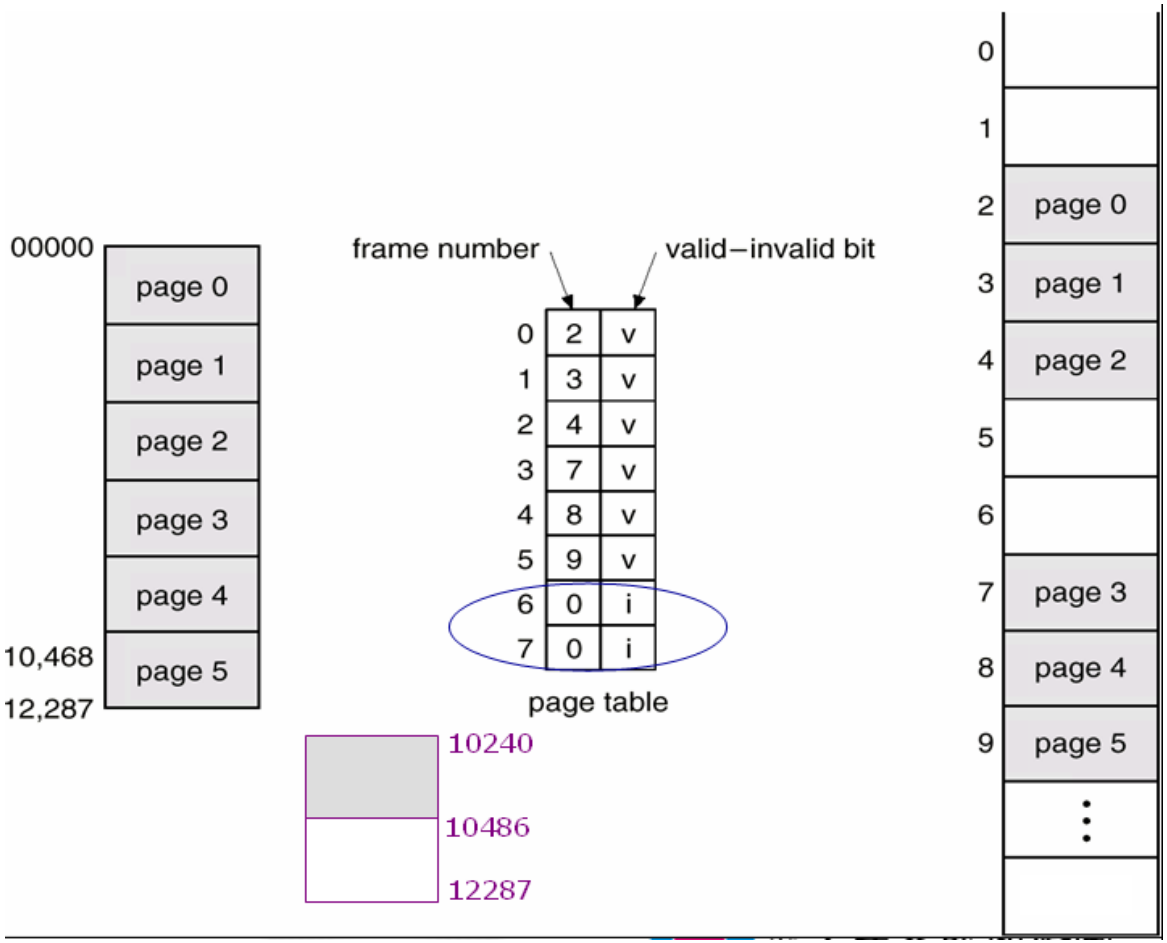
Outline

Memory Protection

- If page size 2^n, page & frame is aligned at 2^n, so ···
- Memory protection implemented by associating **protection bit** with each frame
  - Provide read only, read-write, execute-only protection or···
  - **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

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

- Address space 2^{14}, Page size 2KB; Process size (0~10468):
- Page 5 has internal fragmentation
- PTLR=6, Page 6 & 7 are invalid

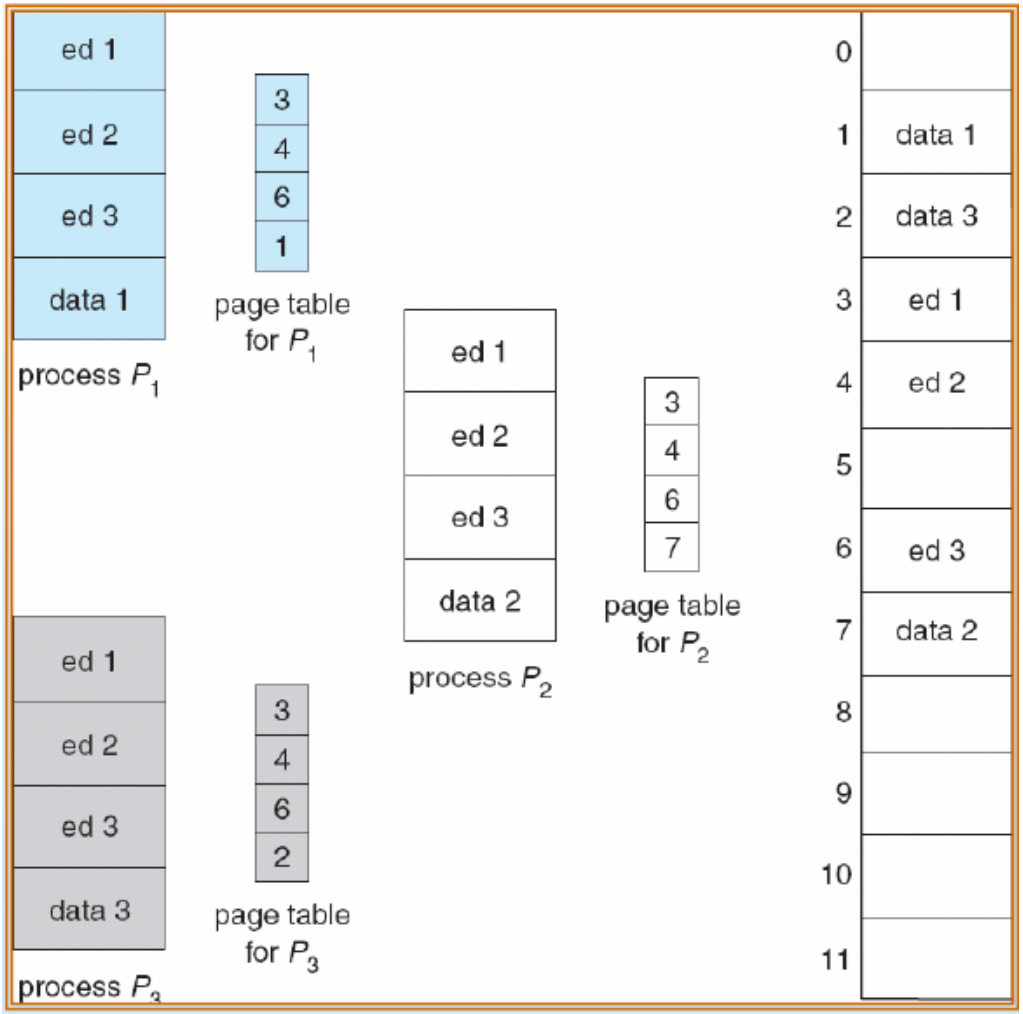


Outline

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 **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

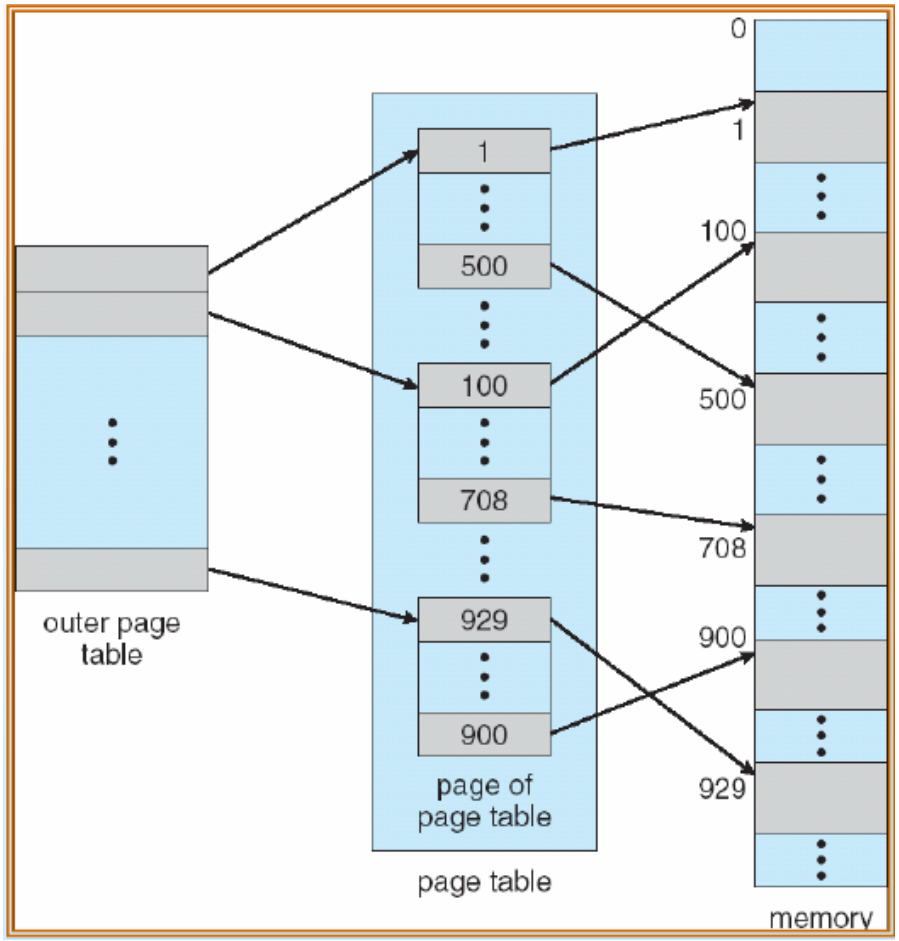


Outline

Hierarchical Page Tables

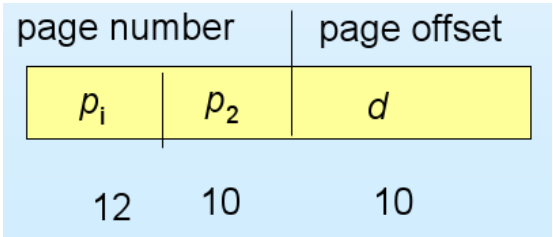
- Break up the logical address space into multiple page tables
  - need directories
- A simple technique is a two-level page table

Two-Level Page-Table Scheme



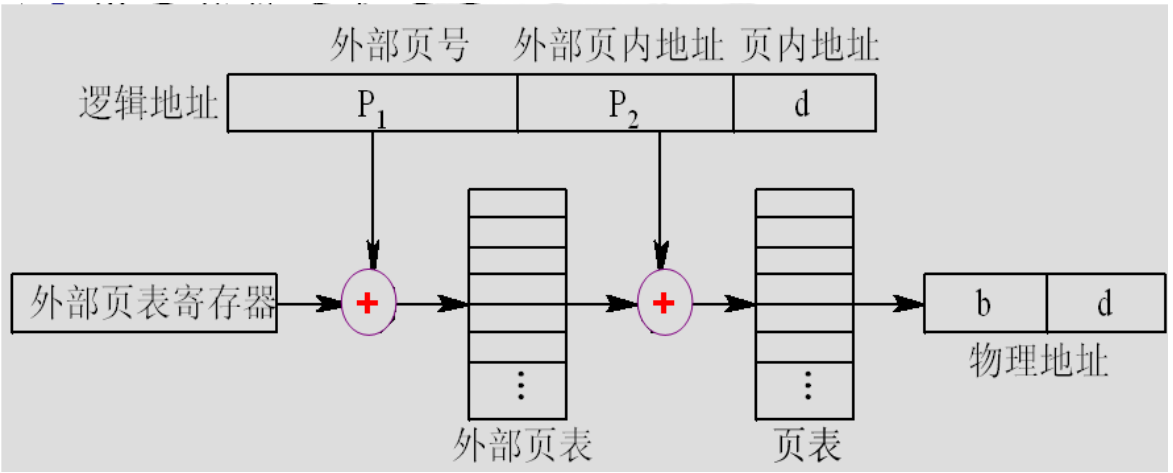
Two-Level Paging Example

- A logical address (on **32-bit** machine with **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

Address-Translation Scheme



Three-level Paging Scheme

outer page	inner page	offset
p_1	p_2	d
42	10	12

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

performance of multi-level page tables

- *Level number* $= L$,
  $EAT = (L+1)t$
- 考虑高速缓存技术：

$$
\begin{aligned}
EAT &= \alpha\,(t+\epsilon) + (1-\alpha)\,((L+1)\,t+\epsilon) \\
\alpha &= 0.98 \\
L &= 3
\end{aligned}
$$

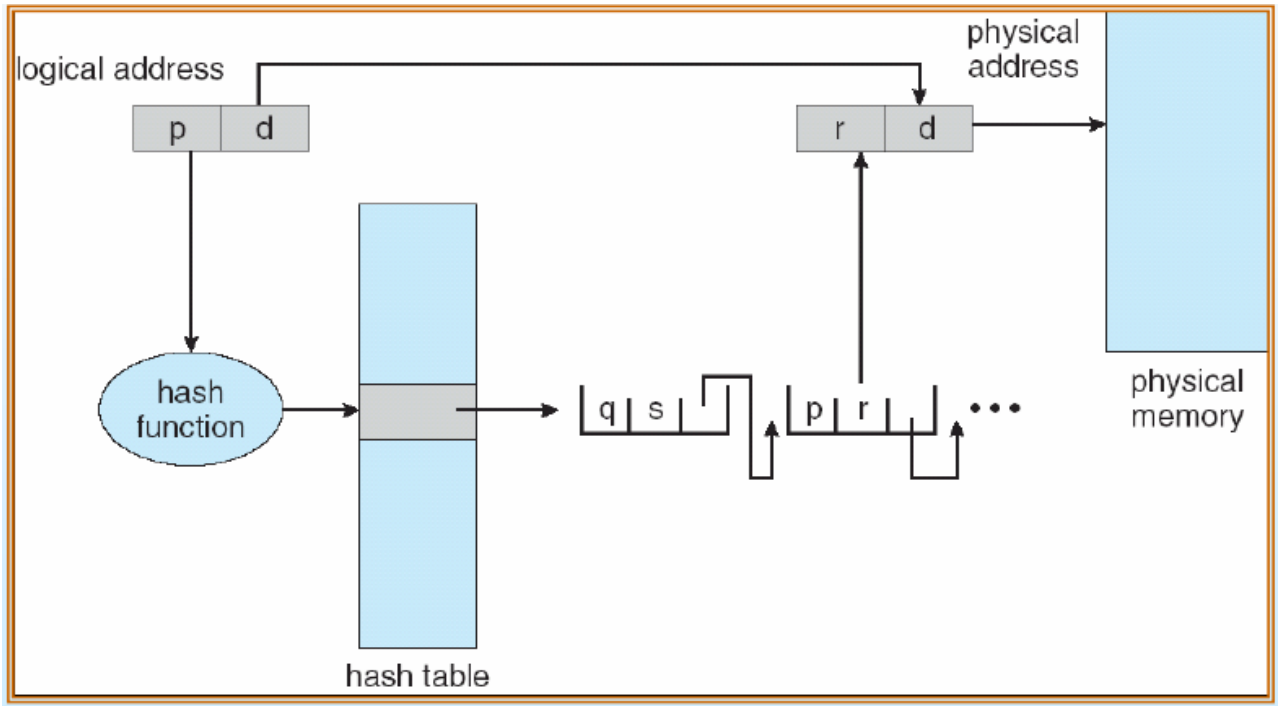
$$
\begin{aligned}
EAT &= 0.98 \times 120 + 0.02 \times 520 \\
&= 128ns
\end{aligned}
$$

which is only a 28% slowdown in memory access time.

Outline

Hashed Page Tables

- Common in address spaces $>$ 32 bits
- The virtual page number is hashed into a page table. This page table 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.

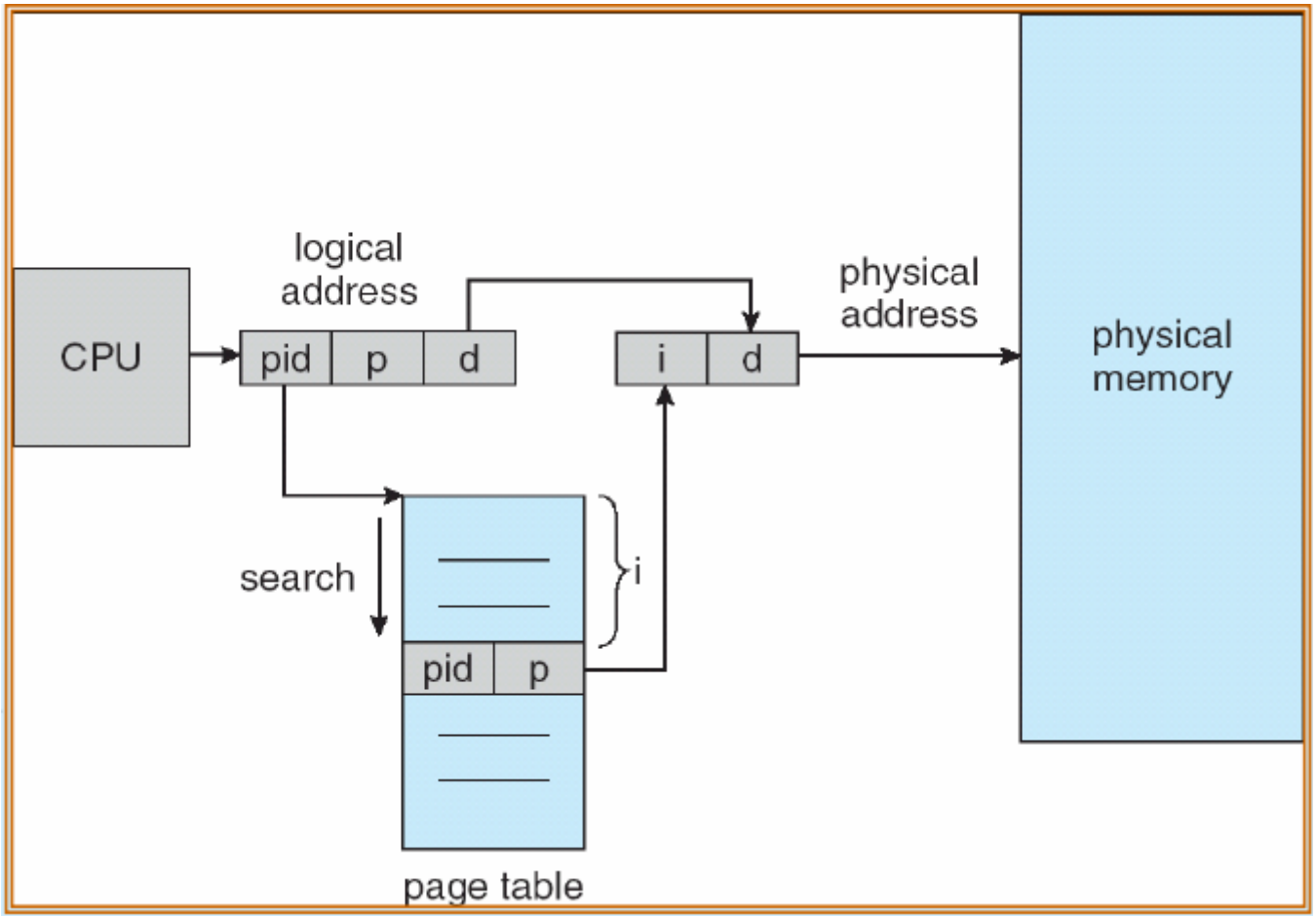


Outline

Inverted Page Table

- One entry for each real page 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
- Use hash table to limit the search to one —or at most a few —page-table entries

Inverted Page Table Architecture



小结

1. Paging (分页)
   - Basic Method
   - Hardware support
   - Memory Protection
   - Shared Pages
2. Structure of the Page Table
   - Hierarchical Paging
   - Hashed Page Tables
   - Inverted Page Tables

谢谢！