



# **Operating Systems**

## **Main Memory-Part2**

Seyyed Ahmad Javadi

[sajavadi@aut.ac.ir](mailto:sajavadi@aut.ac.ir)

Fall 2021

# 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.
- First fit analysis reveals that given  $N$  blocks allocated, another  $0.5 * N$  blocks lost to fragmentation
  - $1/3$  may be unusable -> **50-percent rule**



# Fragmentation (Cont.)

---

- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time
- The simplest compaction algorithm is to move all processes toward one end of memory; all holes move in the other direction, producing one large hole of available memory. This scheme can be expensive.
- Now consider that backing store has same fragmentation problems.



# Paging

---

- Physical address space of a process can be **noncontiguous**;  
process is allocated physical memory whenever **the latter** is available
  - **Avoids external fragmentation**
  - Avoids problem of varying sized memory chunks
- Divide physical memory into fixed-sized blocks called **frames**
  - Size is power of 2, between 512 bytes and 16 Mbytes
- Divide logical memory into blocks of same size called **pages**



# Paging (Cont.)

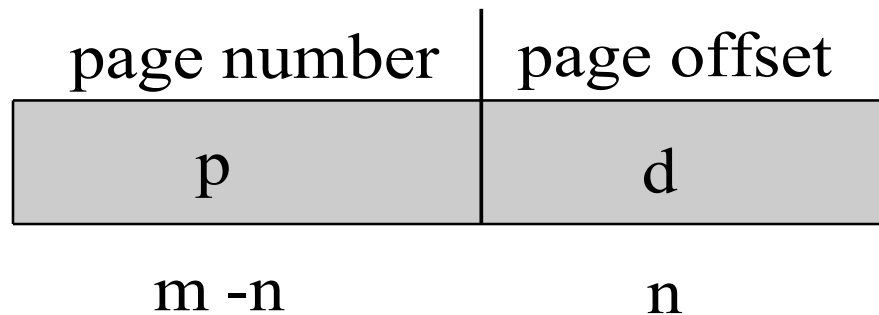
---

- 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
- **Backing store** likewise split into **pages**
- **Still have Internal fragmentation**



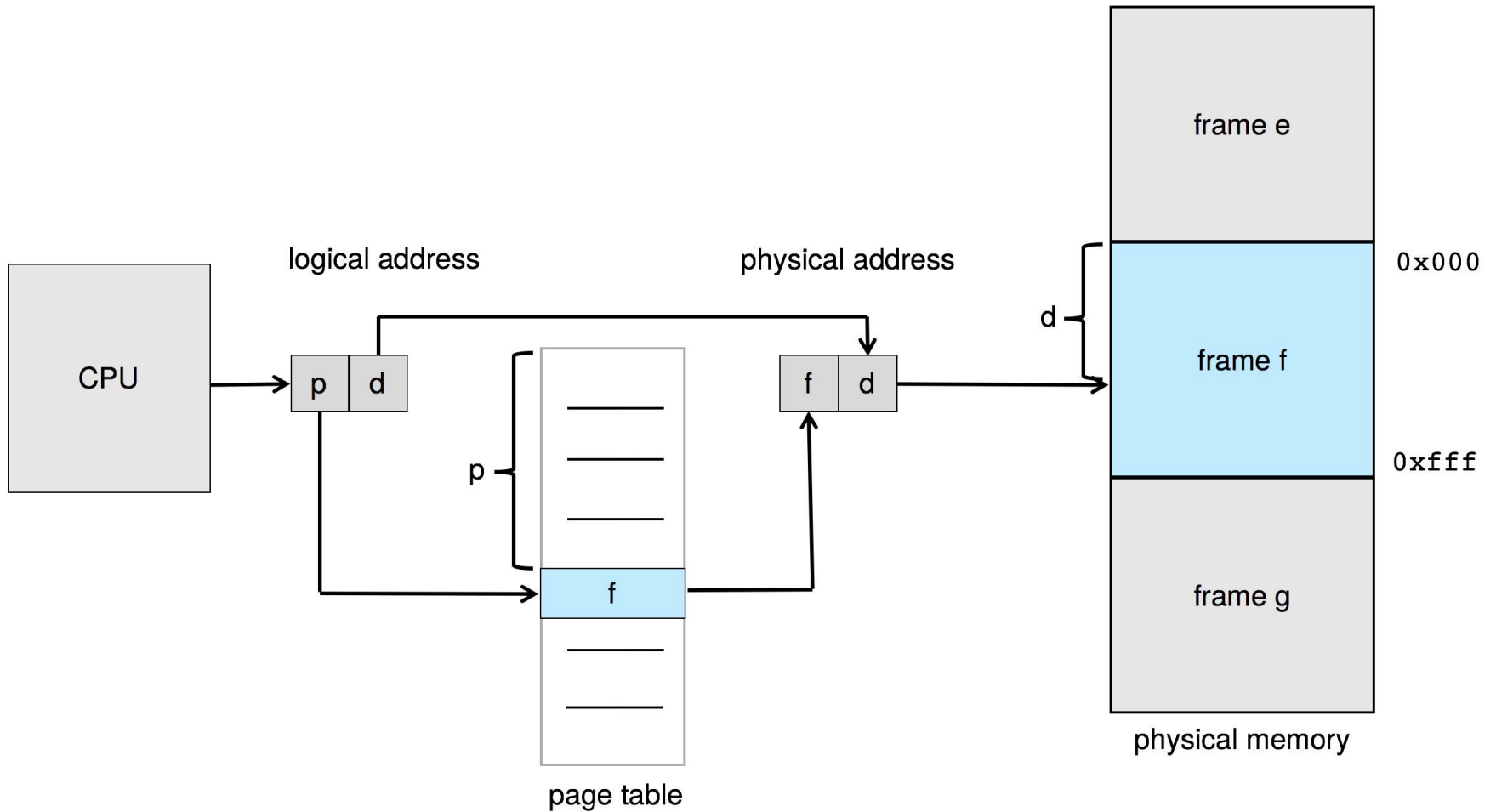
# Address Translation Scheme

- Address generated by CPU is divided into:
  - Page number ( $p$ ) – used as an index into a page table which contains base address of each page in physical memory
  - Page offset ( $d$ ) – combined with base address to define the physical memory address that is sent to the memory unit

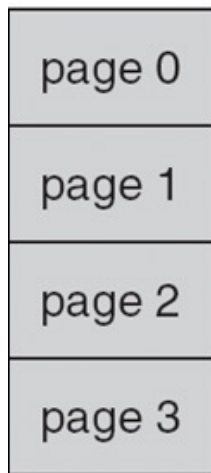


- For given logical address space  $2^m$  and page size  $2^n$

# Paging Hardware



# Paging Model of Logical and Physical Memory

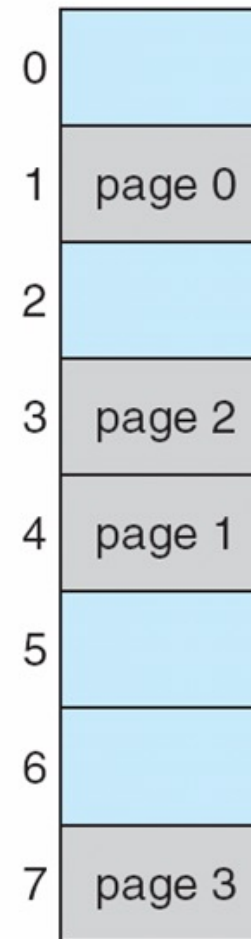


logical  
memory

0	1
1	4
2	3
3	7

page table

frame  
number

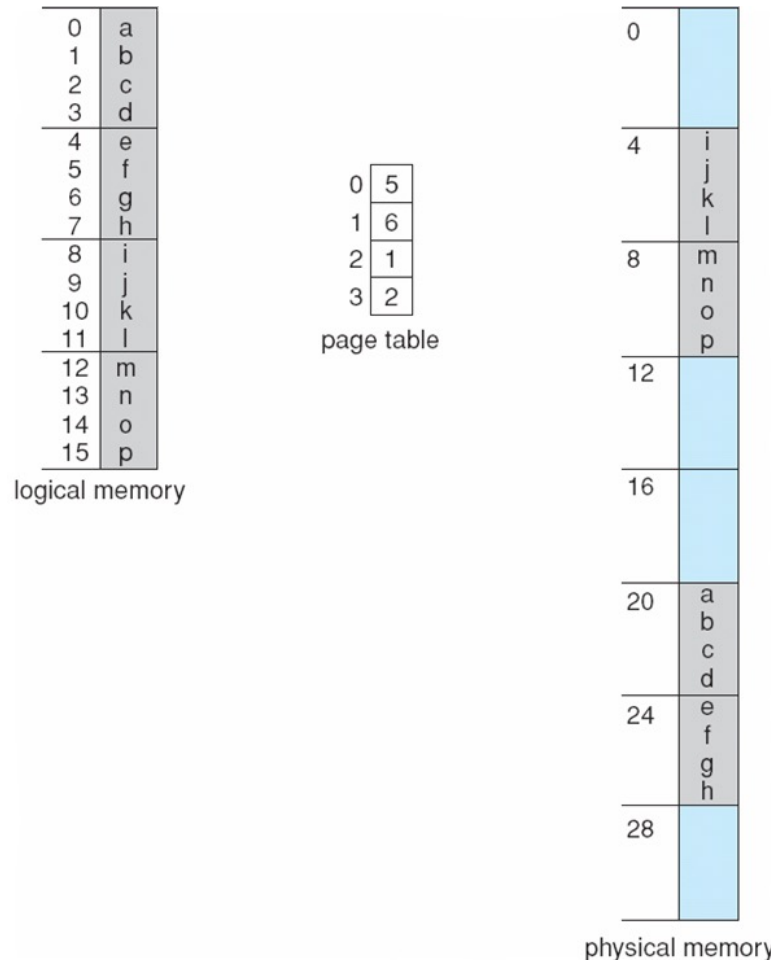


physical  
memory



# Paging Example

- Logical address:  $n = 2$  and  $m = 4$ . Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages)



# Paging -- Calculating internal fragmentation

---

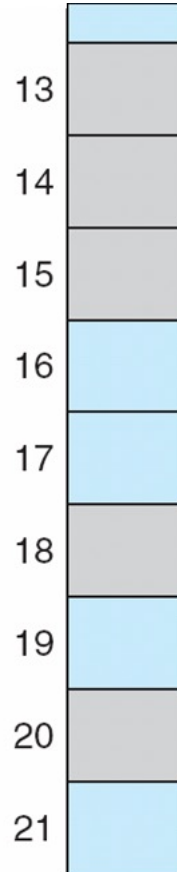
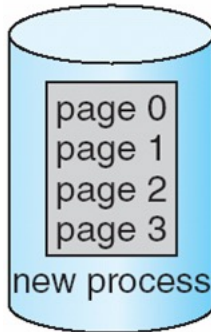
- Page size = 2,048 bytes
- Process size = 72,766 bytes
- 35 pages + 1,086 bytes
- Internal fragmentation of  $2,048 - 1,086 = 962$  bytes
- Worst case fragmentation = 1 frame – 1 byte
- On average fragmentation =  $1 / 2$  frame size
- So small frame sizes desirable?
- But each page table entry takes memory to track
- Page sizes growing over time
  - Solaris supports two page sizes: 8 KB and 4 MB



# Free Frames

free-frame list

14  
13  
18  
20  
15

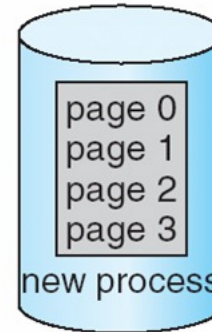


(a)

Before allocation

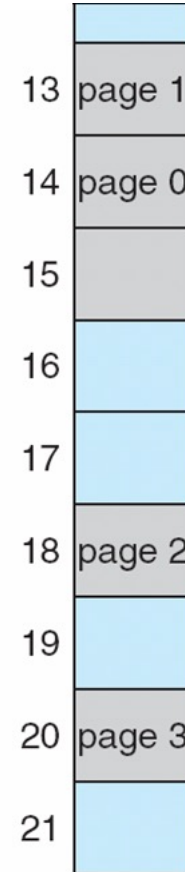
free-frame list

15



0	14
1	13
2	18
3	20

new-process page table



(b)

After allocation

# Implementation of Page Table

---

- Page table is kept in main memory
  - **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 **translation look-aside buffers (TLBs)** (also called **associative memory**).



# Translation Look-Aside Buffer

---

- Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry – uniquely identifies each process to provide address-space protection for that process.
  - Otherwise need to flush at every context switch.
- TLBs typically small (64 to 1,024 entries)
- On a TLB miss, value is loaded into the TLB for faster access next time
  - Replacement policies must be considered
  - Some entries can be **wired down** for permanent fast access



# Hardware

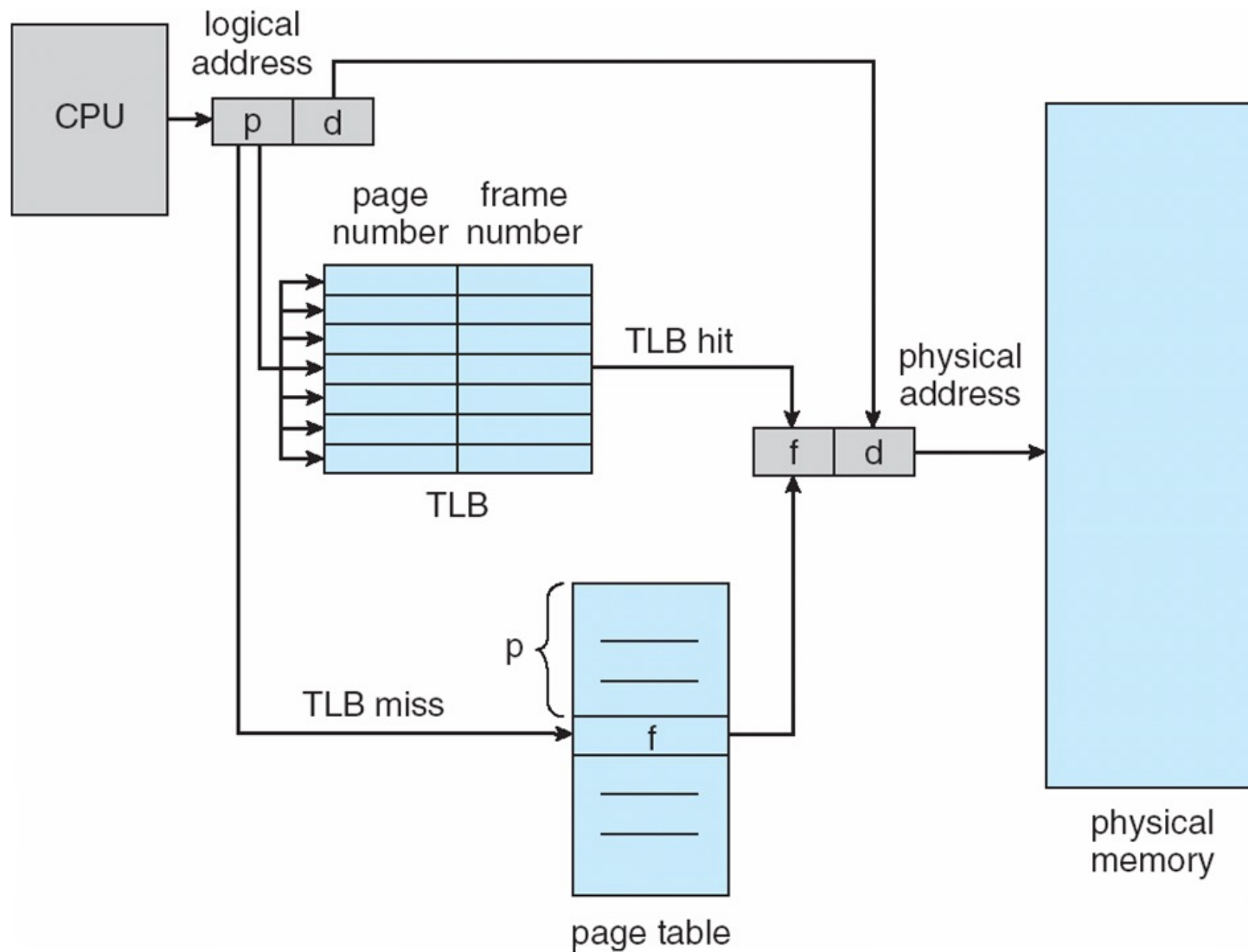
---

## ■ Associative memory – **parallel search**

Page #	Frame #

- Address translation (p, d)
  - If p is in associative register, get frame # out
  - Otherwise get frame # from page table in memory

# Paging Hardware With TLB



# Effective Access Time

---

- Hit ratio: percentage of times that a page number is found in the TLB
- An 80% hit ratio means that we find the desired page number in the TLB 80% of the time.
- Suppose that 10 nanoseconds to access memory.
  - If we find the desired page in TLB then a mapped-memory access take 10 ns.
  - Otherwise, we need two memory access so it is 20 ns





# Effective Access Time (Cont.)

---

- **Effective Access Time (EAT)**

$$\text{EAT} = 0.80 \times 10 + 0.20 \times 20 = 12 \text{ nanoseconds}$$

implying 20% slowdown in access time

- Consider a more realistic hit ratio of 99%,

$$\text{EAT} = 0.99 \times 10 + 0.01 \times 20 = 10.1\text{ns}$$

implying only 1% slowdown in access time.



# 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

00000

page 0
page 1
page 2
page 3
page 4
page 5

10,468  
12,287

frame number      valid-invalid bit

0	2	v
1	3	v
2	4	v
3	7	v
4	8	v
5	9	v
6	0	i
7	0	i

page table

0	
1	
2	page 0
3	page 1
4	page 2
5	
6	
7	page 3
8	page 4
9	page 5
	⋮
	page <i>n</i>

# Shared Pages

---

## ■ Shared code

- One copy of read-only (**reentrant**) code shared among processes (i.e., text editors, compilers, window systems)
- Similar to multiple threads sharing the same process space
- Also useful for interprocess communication if sharing of read-write pages is allowed.

## ■ 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

