# CSCI315 – Operating Systems Design

Department of Computer Science

Bucknell University

## Memory Management: Paging 2

Ch 9.4 – 9.5

*This set of notes is based on notes from the textbook authors, as well as L. Felipe Perrone, Joshua Stough, and other instructors.*
*Xiannong Meng, Fall 2021.*

## 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 from memory into the TLB for faster access next time
  - Replacement policies must be considered
  - Some entries can be **wired down** for permanent fast access

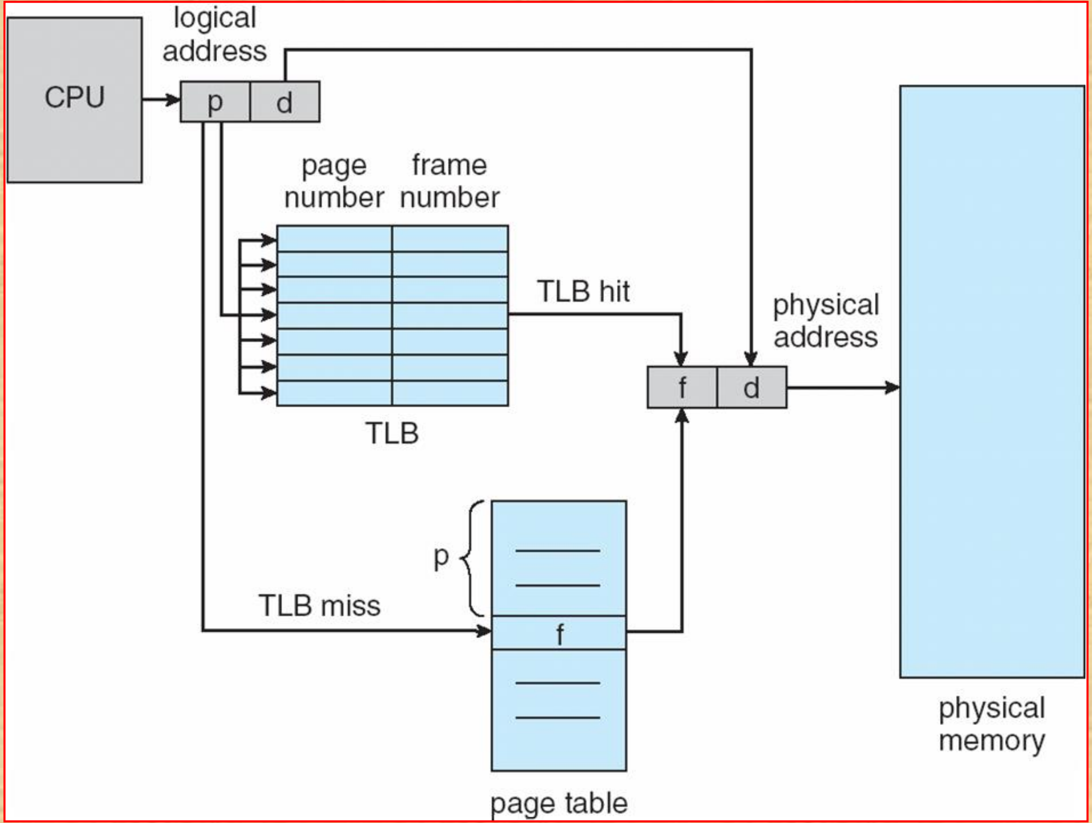
## Hardware

- Associative memory – parallel search

| Page # | Frame # |
|---|---|
| | |
| | |
| | |
| | |

- Address translation (p, d)
  - If **p** is in associative register, retrieve the frame # **f** from TLB
  - Otherwise get frame # from page table in memory

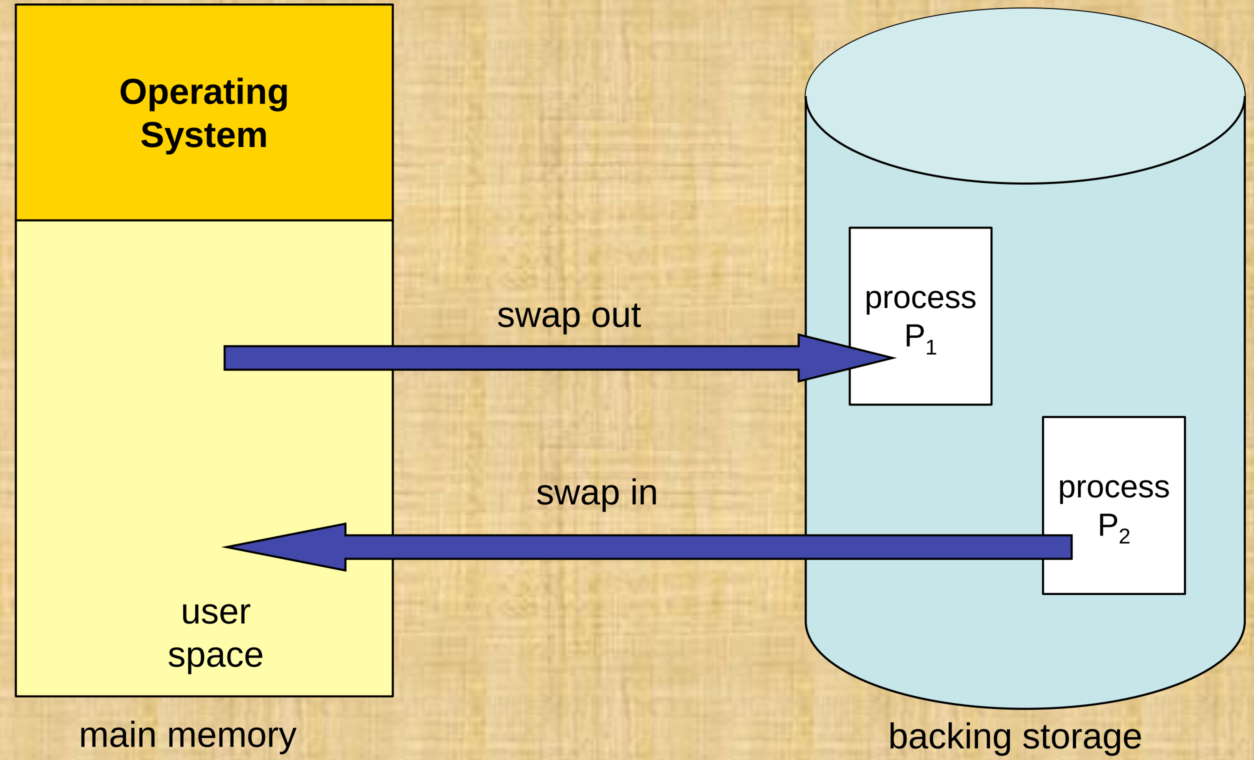
# Paging Hardware With TLB



## 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** – fast disk 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.
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped.
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows).

# Schematic View of Swapping

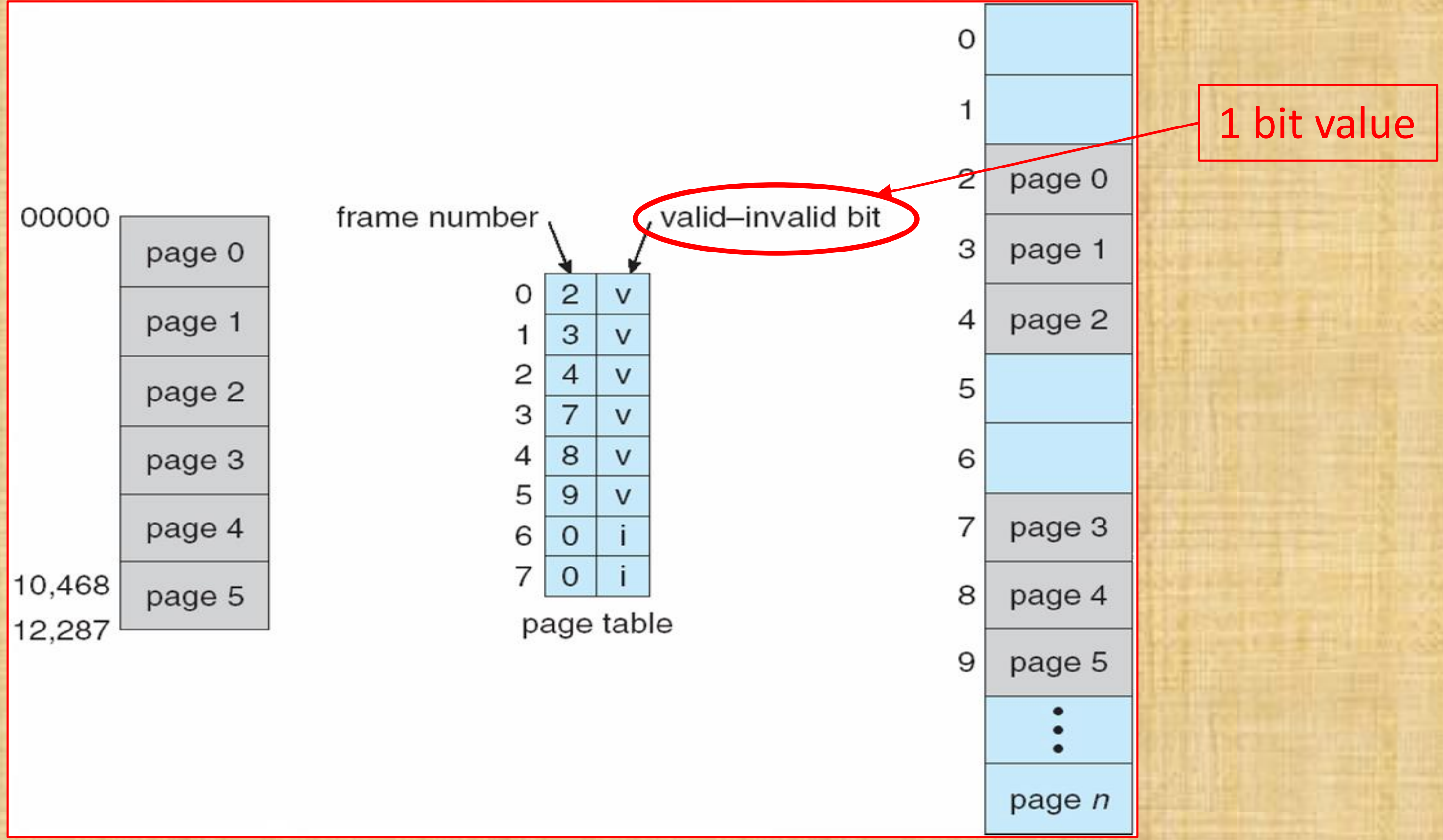


# Memory Protection

- Memory protection implemented by associating protection bits with each frame.

- *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 Bit



|  | 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 |

# Effective Access Time

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

why 2?

$$EAT = (\mathbf{t} + \varepsilon)\,\alpha + (2\mathbf{t} + \varepsilon)(1 - \alpha)$$

# Effective Access Time

- **Effective Access Time (EAT)**

why 2?

$$EAT = (\mathbf{t} + \varepsilon)\,\alpha + (\boxed{2\mathbf{t}} + \varepsilon)(1 - \alpha)$$

- First access to memory is to load the page table entry when it is not in the TLB (cache). The second access actually reads the data or instruction from the memory.

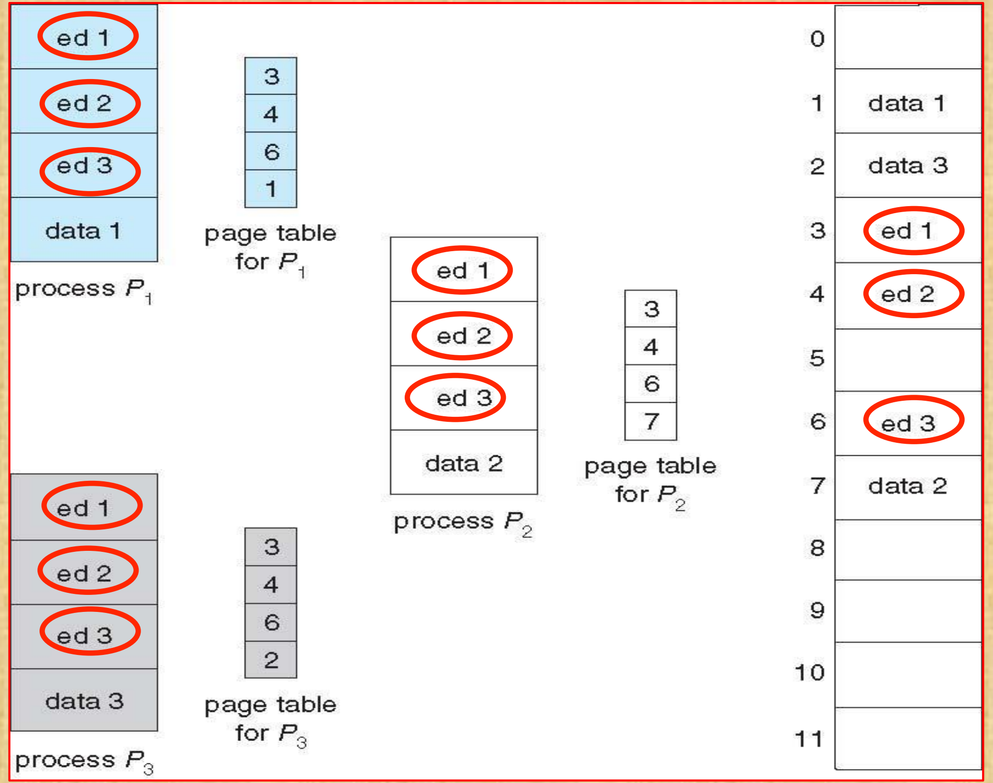
## Effective Access Time Example

- **Effective Access Time (EAT)**

  $$EAT = (\mathbf{t} + \varepsilon)\, \alpha + (2\mathbf{t} + \varepsilon)(1 - \alpha)$$

- Assume $\alpha = 0.8$ (80 percent hit ratio for PTL entries)
- $\varepsilon = 1$ ns, $t = 20$ ns
- EAT = (20+1)*0.80 + (2*20 +1)*0.20 = 25 ns
- If $\alpha = 0.99$, EAT = (20+1)*0.99 + (2*20 +1)*0.01 = 21.2 ns

# Added Benefit: shared pages



## Shared Pages

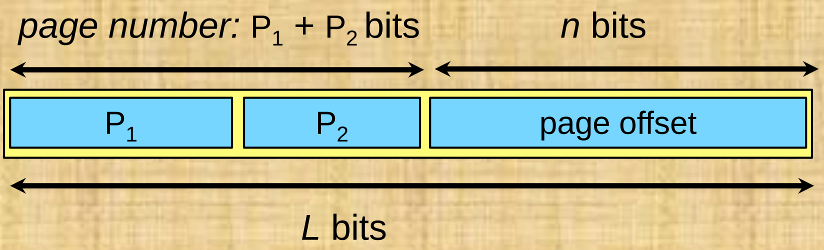
- Shared code
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
  - Shared code must appear in same location in the logical address space of all processes.
- Private code and data
  - Each process keeps a separate copy of the code and data.
  - The pages for the private code and data can appear anywhere in the logical address space.

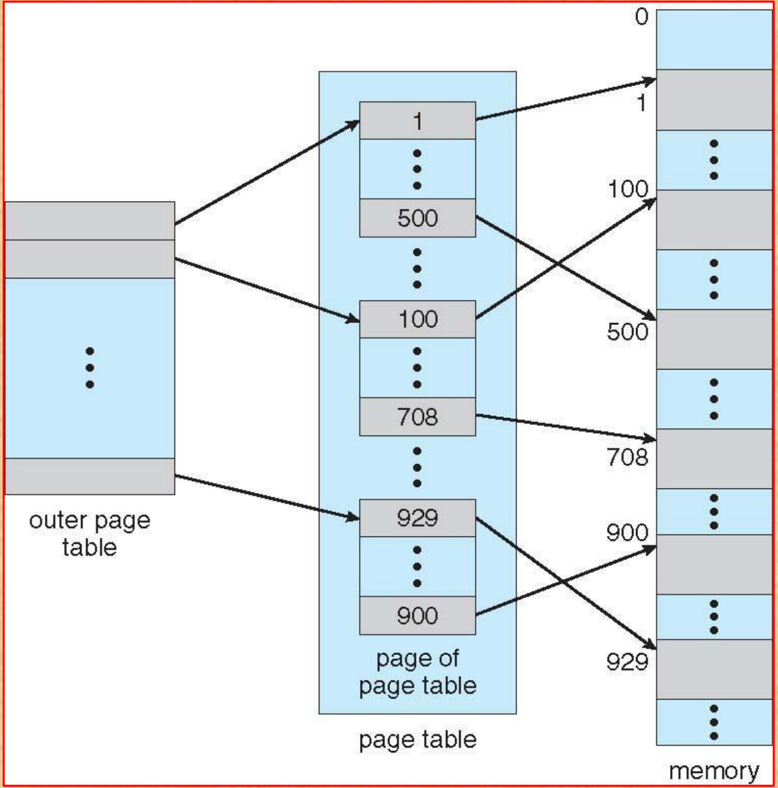
# Large Page Table?

- Break up the logical address space into multiple page tables

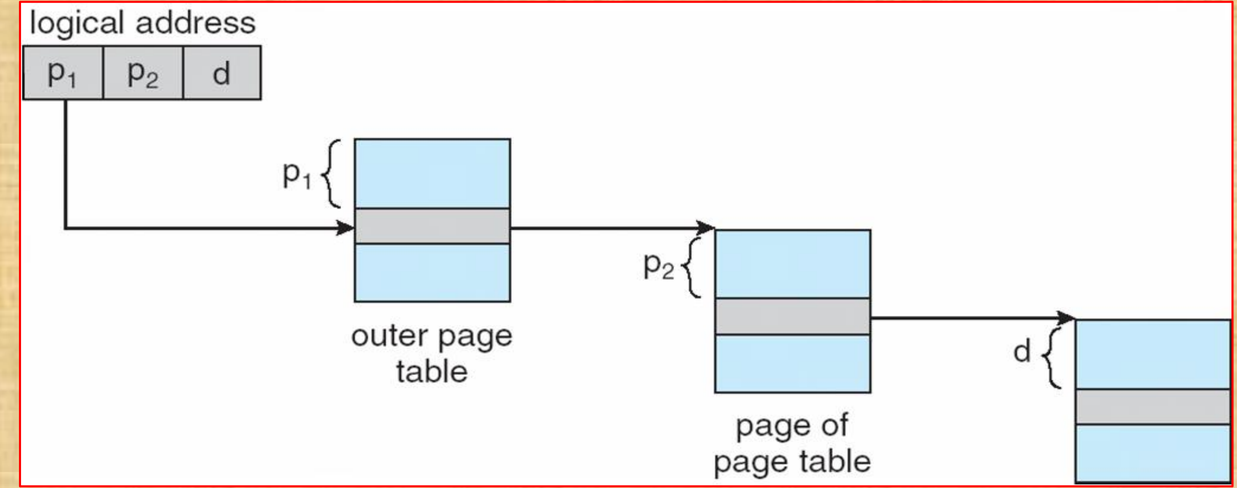
- A simple technique is a two-level page table

- The page table is broken into pages





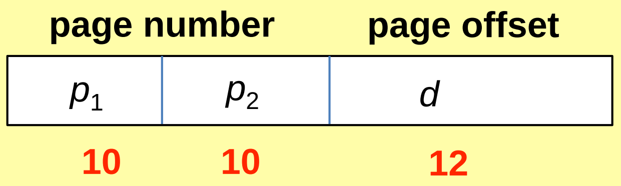
# Address Translation



aggregates all the pages that together make up the page table

# Two-Level Paging Example

- A logical address (on 32-bit machine with 4K page size) is divided into:
  - a page number consisting of 20 bits.
  - a page offset consisting of 12 bits.
- Since the page table is paged, the page number is further divided into:
  - a 10-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 outer page table.