Virtual Memory Management

Didem Unat

Lecture 18

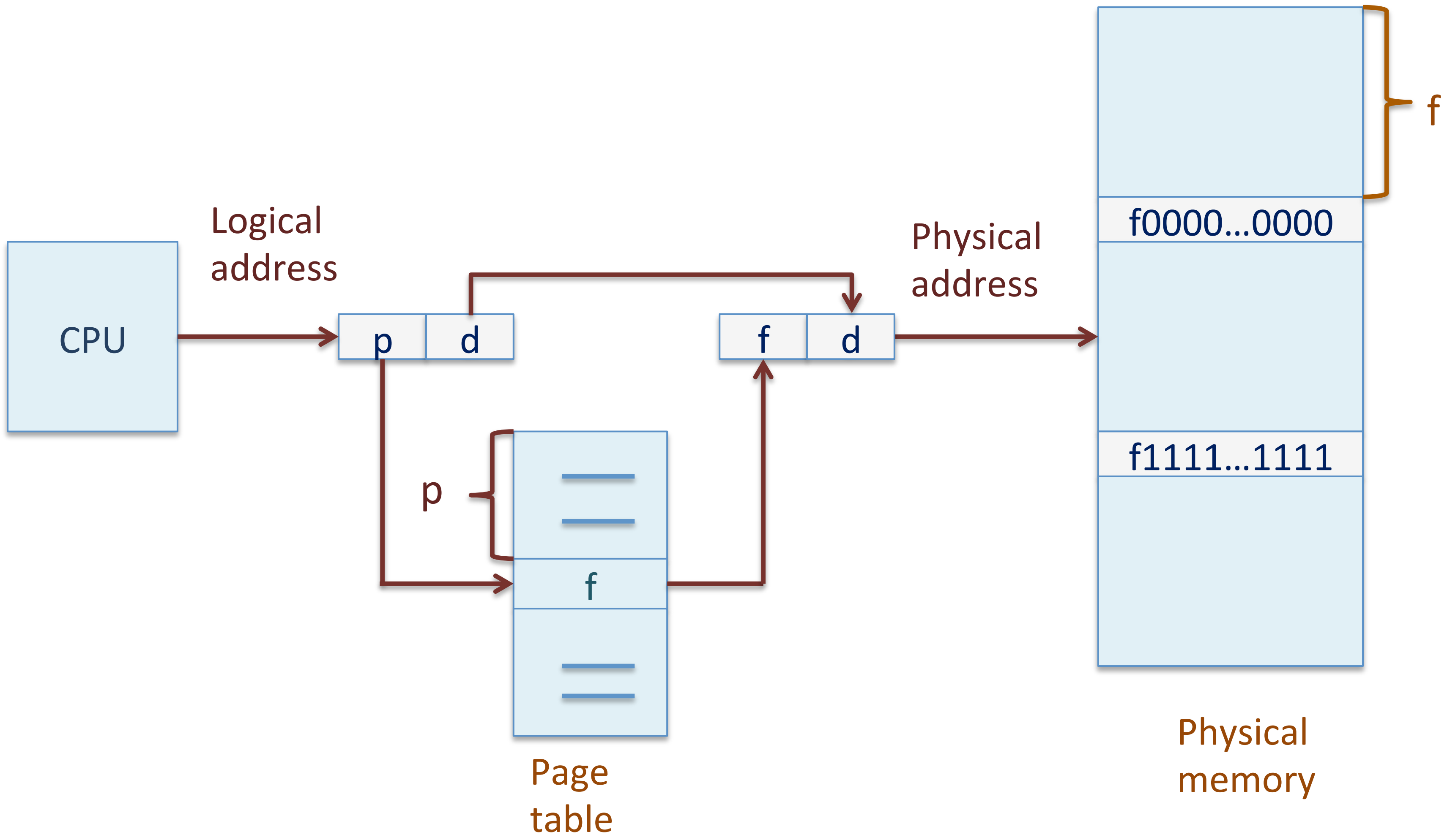
COMP304 - Operating Systems (OS)

Paging

- Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available.
- Divide physical memory into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 16MB).
- Divide logical memory into blocks of the same size called **pages**.

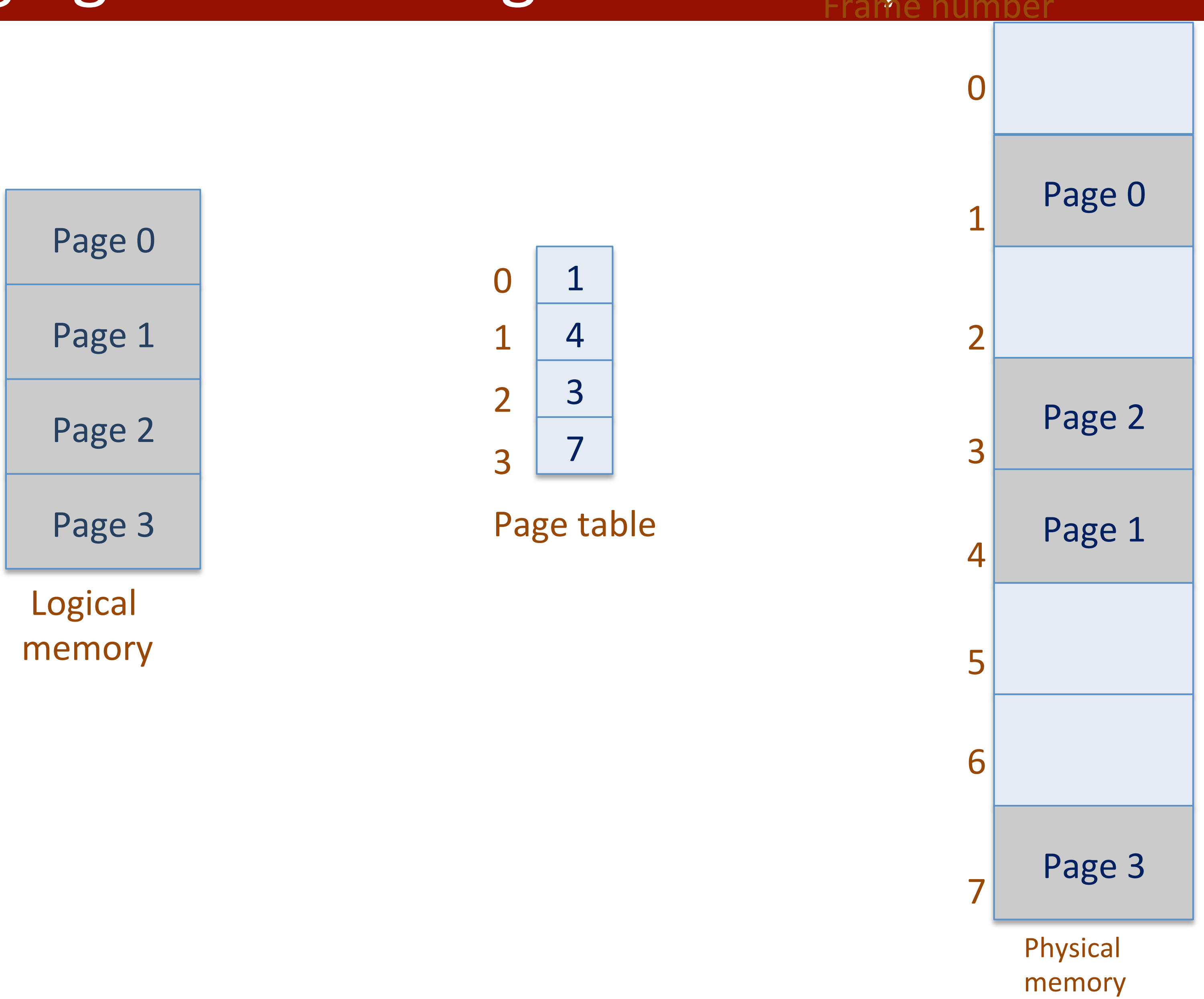
- Keep track of all free frames.
  - Set up a page table to translate logical to physical addresses.

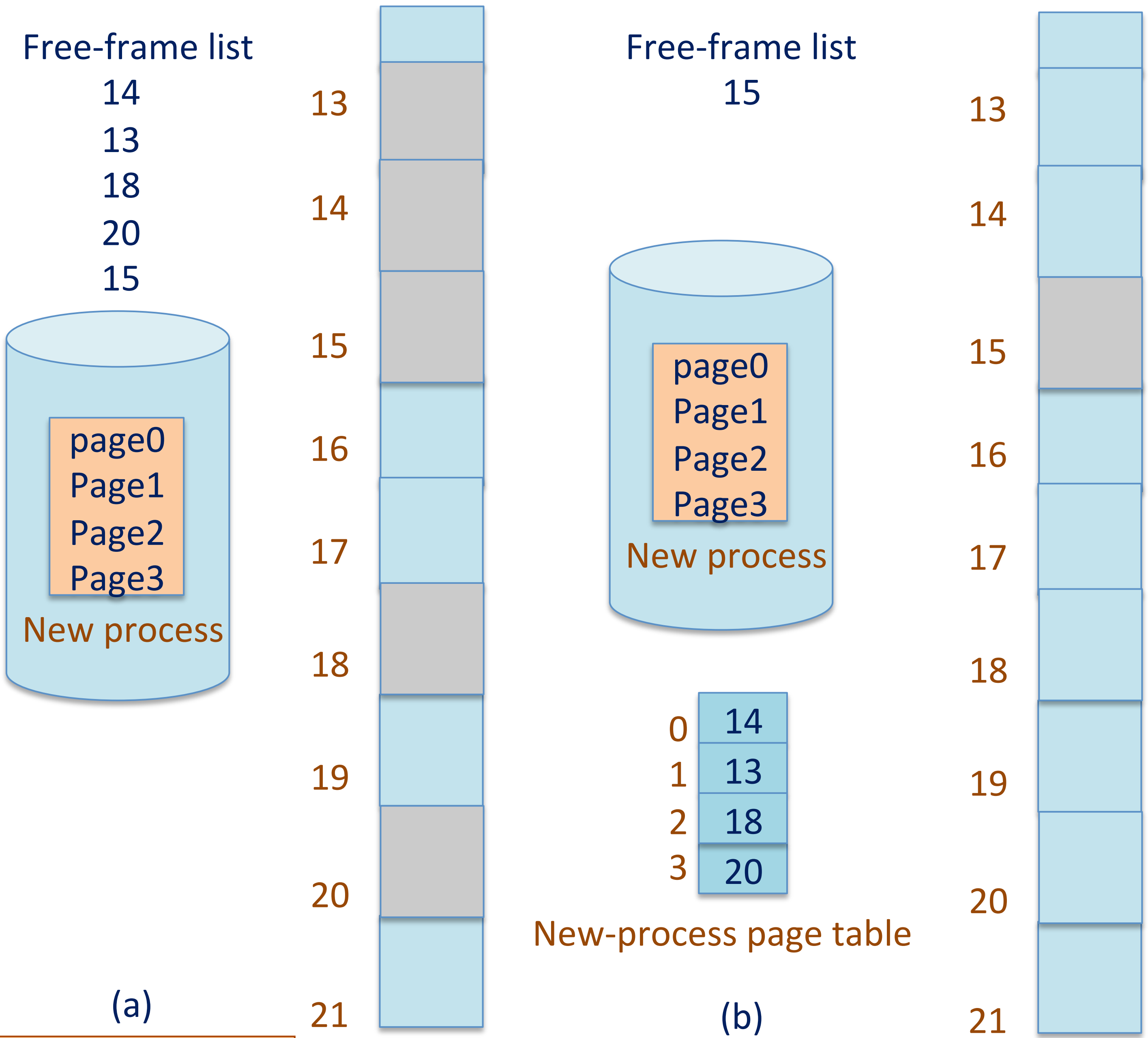
Address Translation Architecture



Paging hardware

Paging Model of Logical and Physical Memory

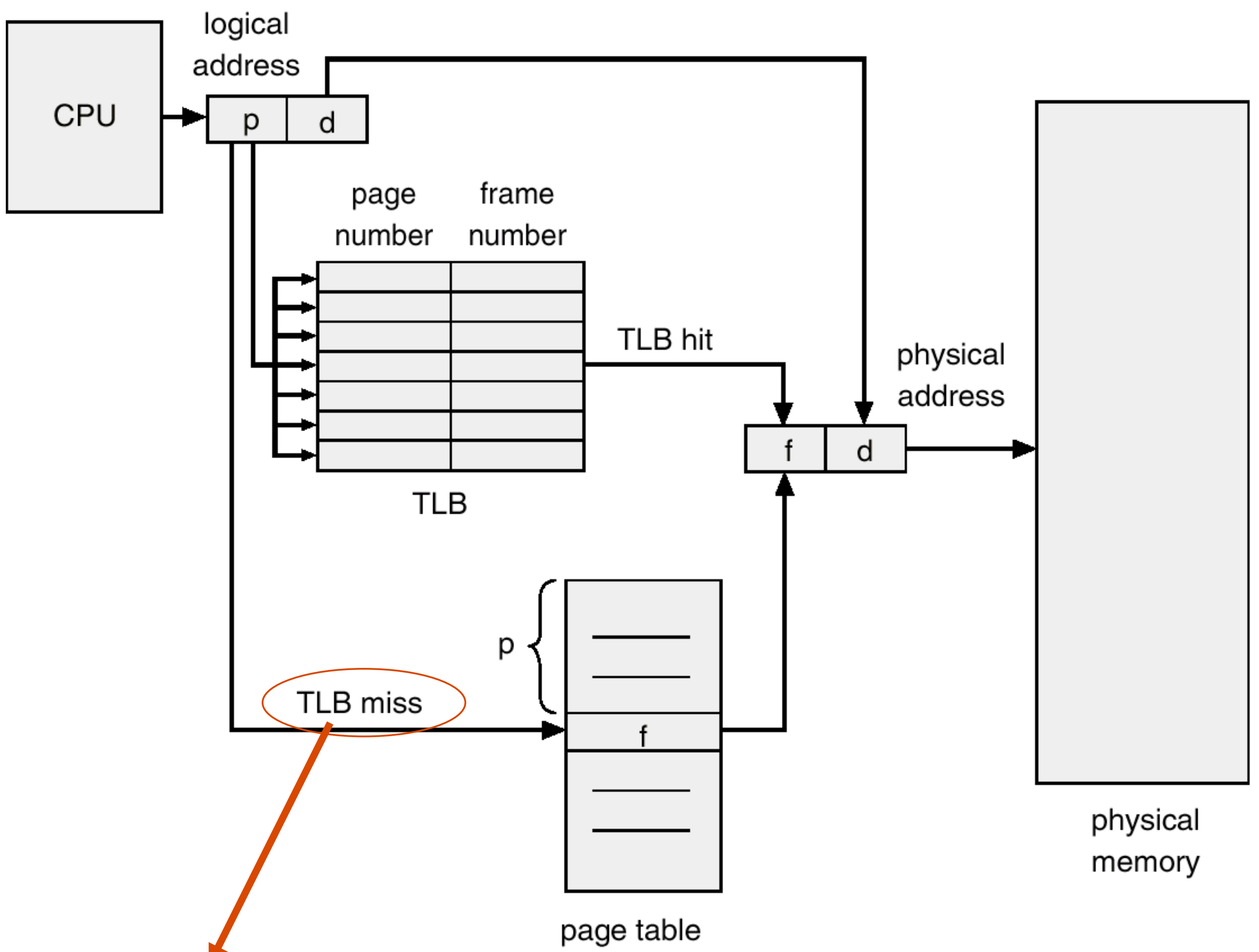




(a) Before allocation

(b) After allocation

Paging Hardware with TLB



Page # and frame # is added to TLB

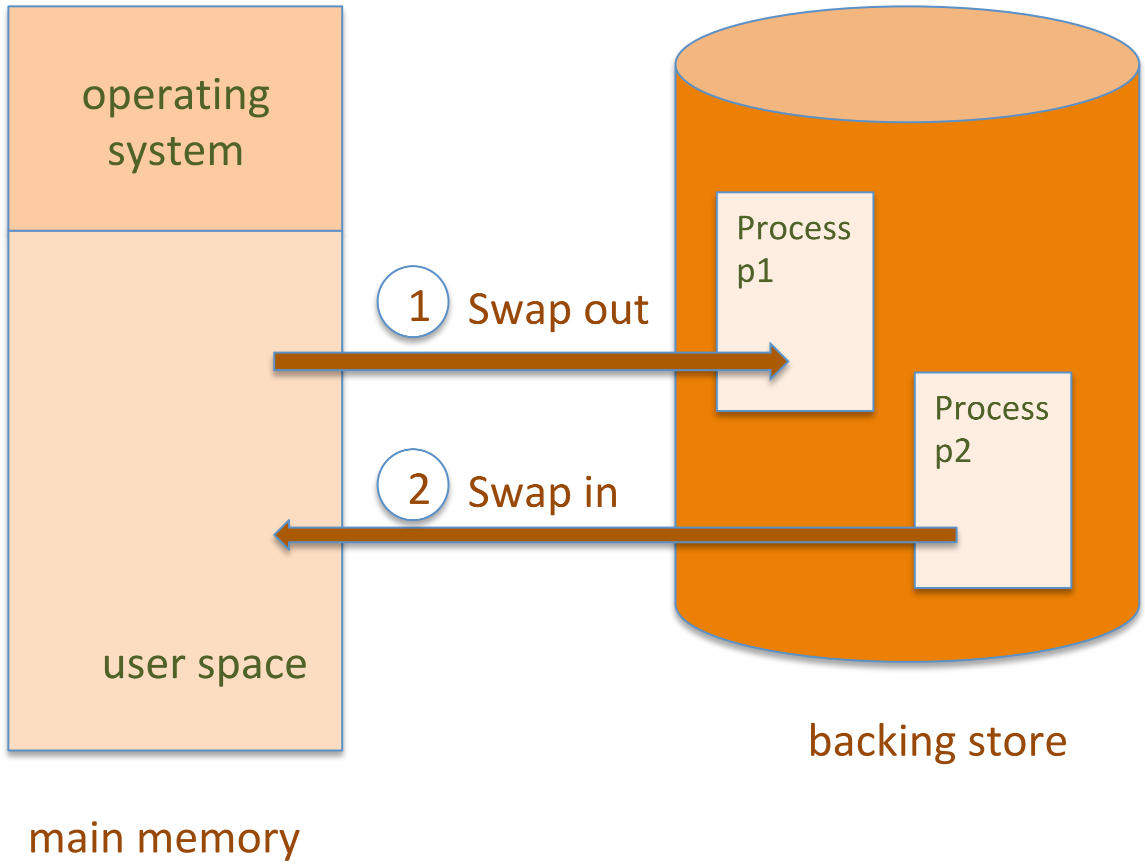
Memory Usage

- **What happens if a process needs more memory than there is available physical memory?**

- **Virtual memory**: separation of user logical memory from physical memory.
  - Only part of the program needs to be in memory for execution.
  - Logical address space can therefore be **much larger than** physical address space.
  - Allows address spaces to be shared by several processes.
  - Allows for more efficient process creation.

Swapping

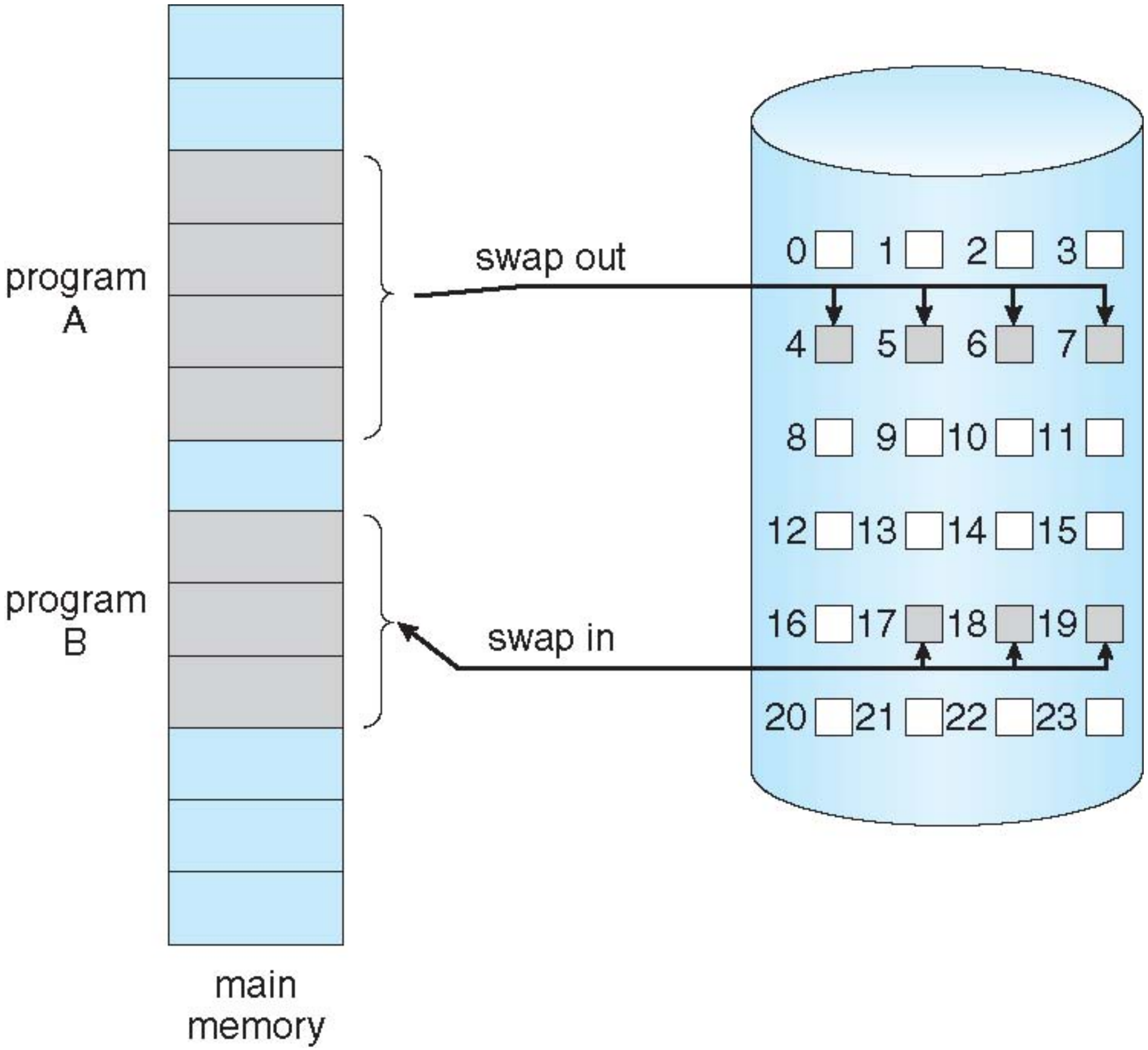
- A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution.



Demand Paging

- Similar to a paging system with swapping
- Bring a page into memory only when it is needed.
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users or processes
- Page is needed $\Rightarrow$ reference to it
  - invalid reference $\Rightarrow$ abort
  - not-in-memory $\Rightarrow$ bring it to memory
- **Lazy swapper** – never swaps a page into memory unless page will be needed

Transfer of a Paged Memory to Contiguous Disk Space



Valid-Invalid Bit

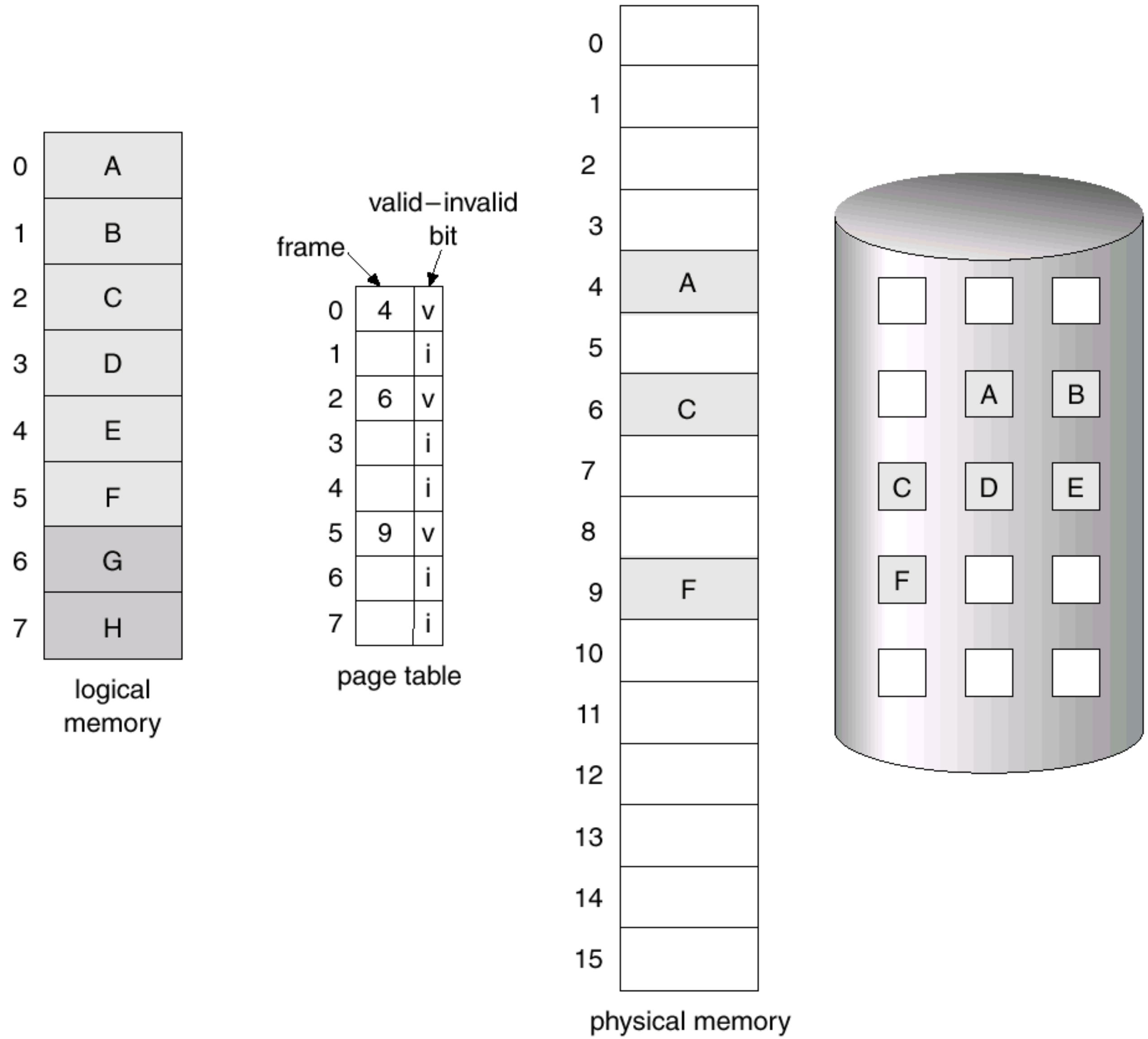
- With each page table entry a valid–invalid bit is associated ($\mathbf{v} \Rightarrow$ in-memory, $\mathbf{i} \Rightarrow$ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:

Frame #	valid-invalid bit
	v
	v
	v
	v
	i
….	
	i
	i

page table

- During address translation, if valid–invalid bit in page table entry is **i** $\Rightarrow$ **page fault**

Page Table when some pages are not in Main Memory



Page Fault

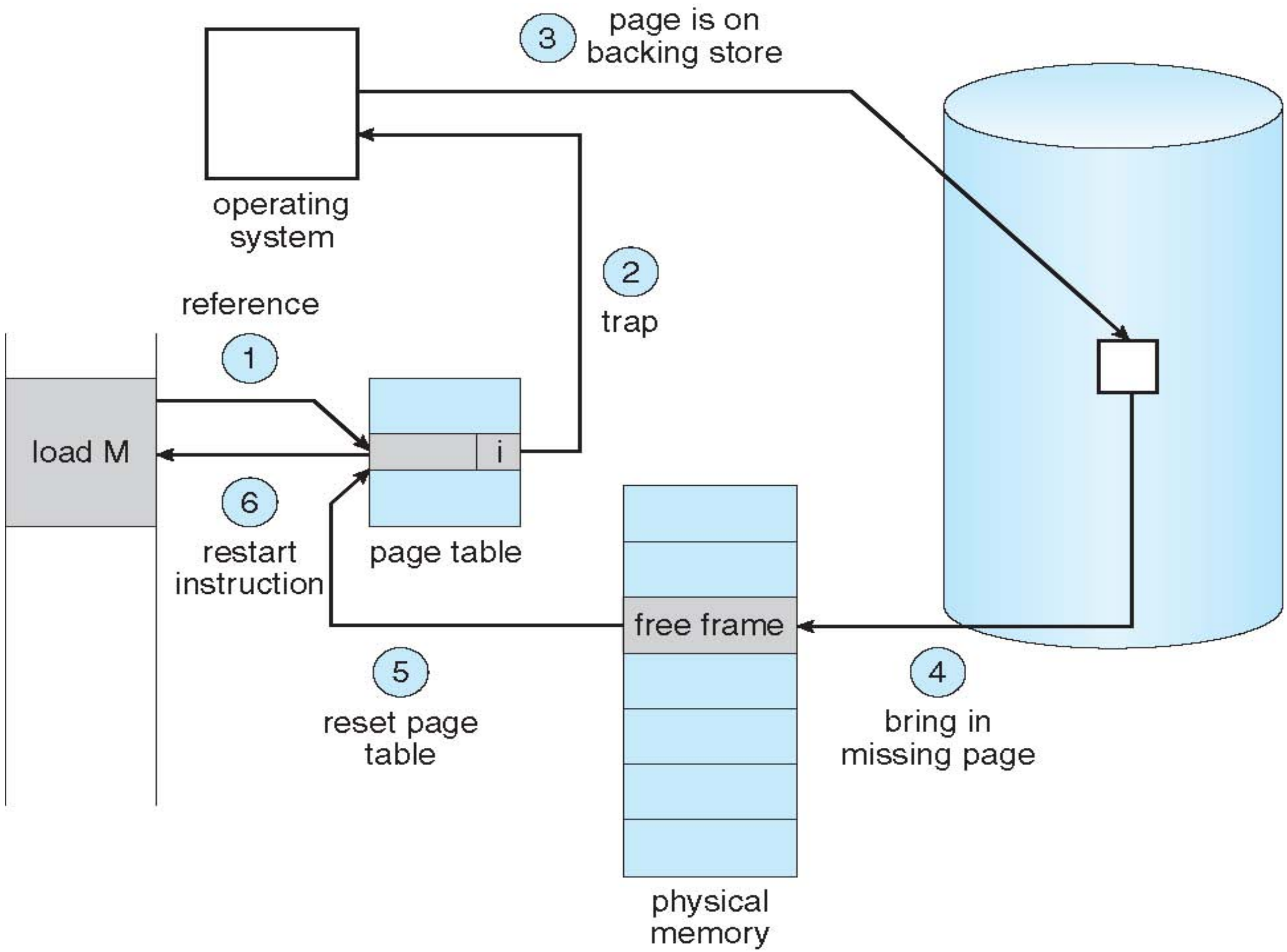
- If there is a reference to a page, first reference to that page will trap to operating system: **page fault**

1. Operating system looks at another table (kept in Process Control Block) to decide:
    - Invalid reference ⇒ abort
        - out of process's allowed address space
    - Just not in memory
        - The page is in the disk
2. Get empty frame
3. Swap page into frame
4. Reset tables
5. Set validation bit = **v**
6. Restart the instruction that caused the page fault

Page Fault (more in detailed)

1. Trap to the operating system
2. Save the user registers and process state
3. Determine that the interrupt was a page fault
4. Check that the page reference was legal and determine the location of the page on the disk
5. Issue a read from the disk to a free frame:
    1. Wait in a queue for this device until the read request is serviced
    2. Wait for the device seek and/or latency time
    3. Begin the transfer of the page to a free frame
6. While waiting, allocate the CPU to some other user
7. Receive an interrupt from the disk I/O subsystem (I/O completed)
8. Save the registers and process state for the other user
9. Determine that the interrupt was from the disk
10. Correct the page table and other tables to show page is now in memory
11. Wait for the CPU to be allocated to this process again
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction

Steps in Handling a Page Fault



Aspects of Demand Paging

- Extreme case – start process with ***no* pages** in memory
  - OS sets instruction pointer to first instruction of a process, non-memory-resident -> page fault
  - And for every other process pages on first access
  - **Pure demand paging**
- Actually, a given instruction could access multiple pages -> multiple page faults
  - How can that happen?
- Hardware support needed for demand paging
  - Page table with valid / invalid bit
  - Secondary memory (swap device with **swap space**)
  - Instruction restart

Question?

- A thread states ***are Ready, Running, and Blocked***,
  - where a thread is either ready and waiting to be scheduled,
  - is running on the processor, or
  - is blocked (for example, waiting for I/O).
- Assuming a thread is in the Running state
  - Will the thread change state if it incurs a page fault?
    - If so, to what new state?
  - Will the thread change state if it generates a TLB miss that is resolved in the page table?
    - If so, to what new state?

- On a page fault the thread state is set to blocked as an I/O operation is required to bring the new page into memory.
- On a TLB-miss, the thread continues running if the address is resolved in the page table.

Performance of Demand Paging

- Page Fault Rate $0 \le p \le 1$
  - if $p = 0$: no page faults
  - if $p = 1$: every reference causes a page fault

- Effective Access Time (EAT)

$$
\begin{aligned}
EAT = (1 - p) &\times \text{memory access} \\
&+ p \times (\text{page fault overhead} \\
&\qquad + \text{swap page out} \\
&\qquad + \text{swap page in} \\
&\qquad + \text{restart overhead})
\end{aligned}
$$

Demand Paging Example

- Memory access time = 200 nanoseconds

- Average page-fault service time = 8 milliseconds

- EAT = (1 – p) x 200 + p x (8 milliseconds)
  = (1 – p) x 200 + p x 8,000,000
  = 200 + p x 7,999,800

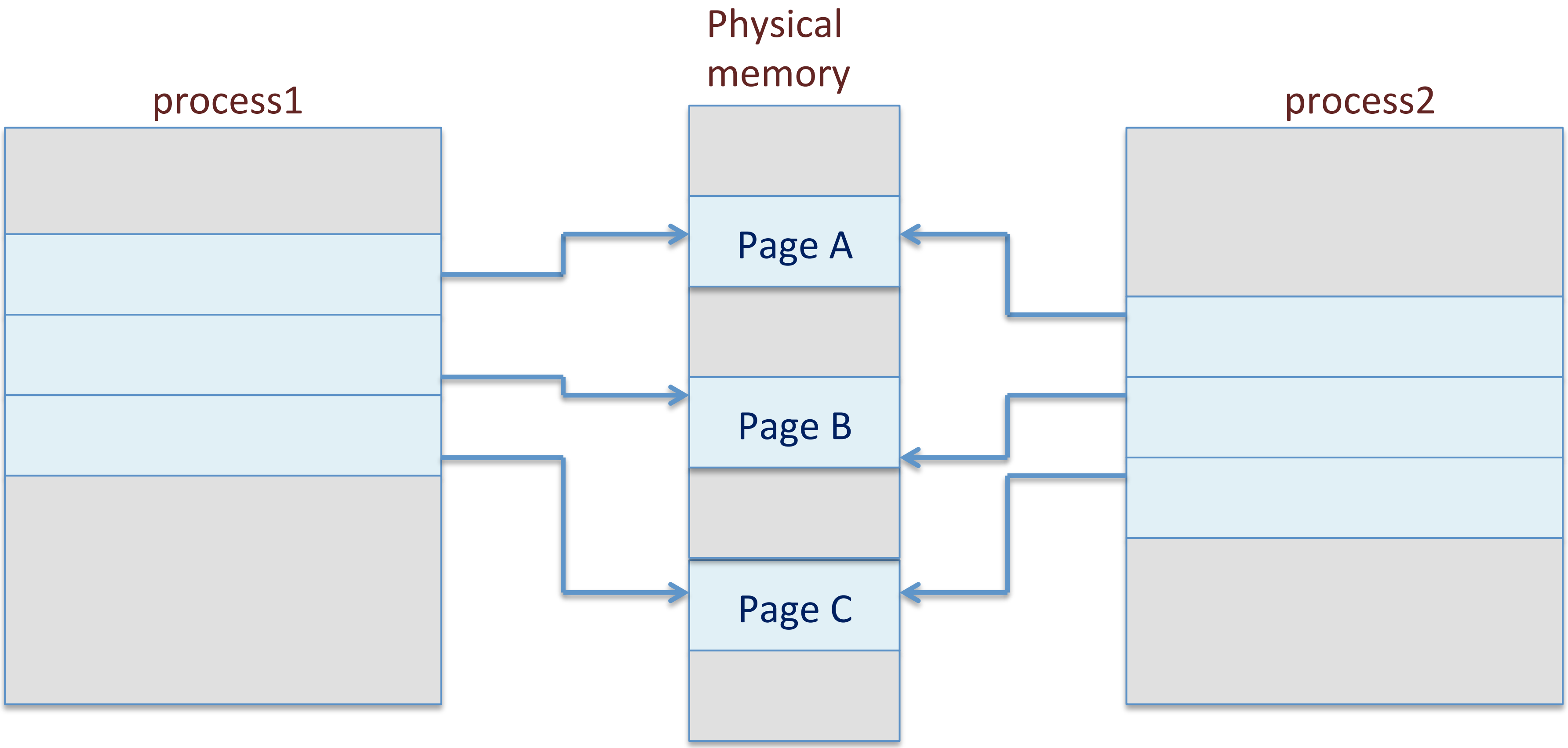
- If one access out of 1,000 causes a page fault, then
  EAT = 8.2 microseconds.
  This is a slowdown by a factor of 40!!

EAT: Effective access time

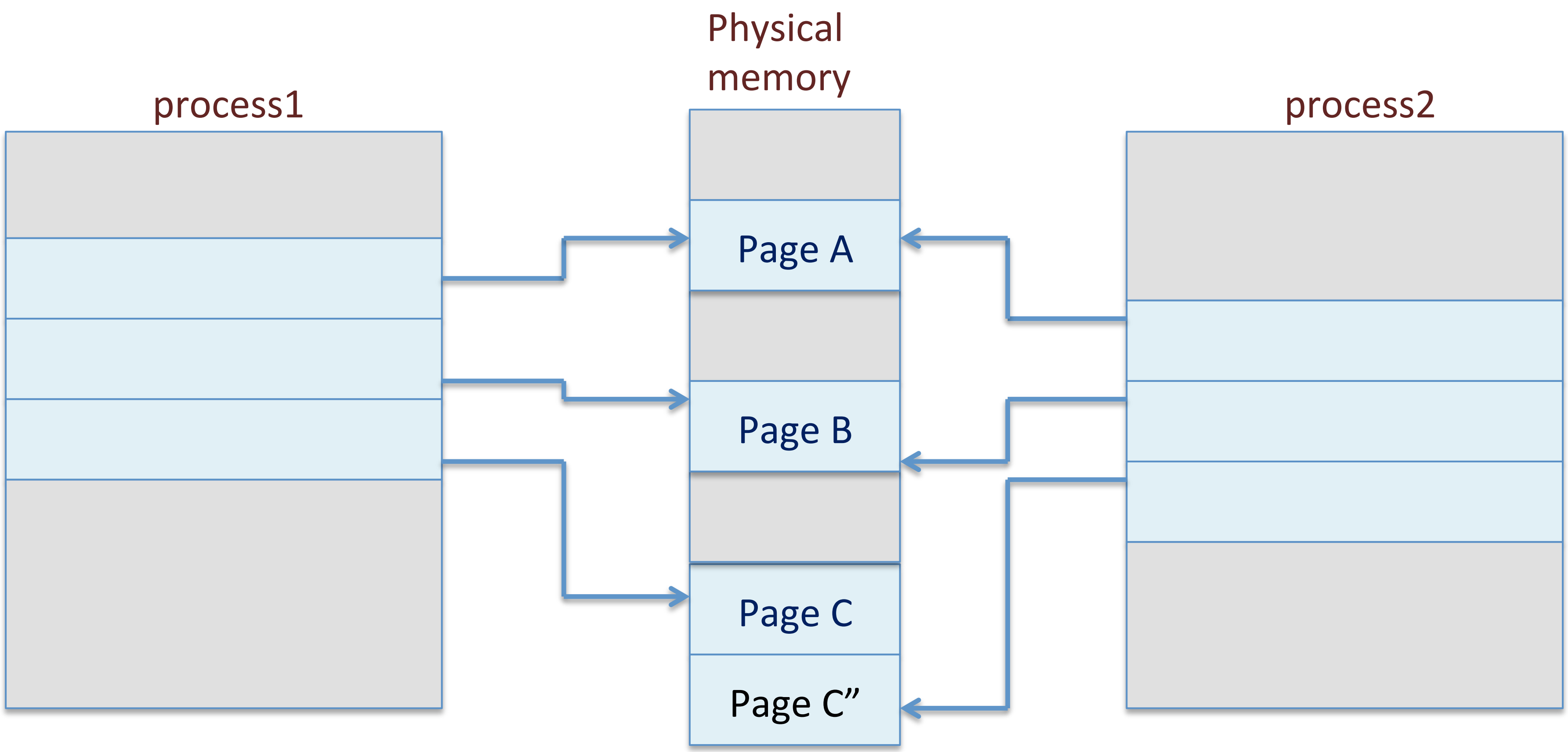
Copy-on-Write

- **Copy-on-Write** (COW) allows both parent and child processes to initially *share* the same pages in memory
  - If either process modifies a shared page, only then is the page copied
- COW allows more efficient process creation as only modified pages are copied
- In general, free pages are allocated from a **pool** of **zero-fill-on-demand** pages

Before Process 2 Modifies Page C



After Process 2 Modifies Page C



What happens if there is no free frame?

- Page replacement – find a frame in memory, but not really in use, swap it out.
  - performance – want an algorithm which will result in minimum number of page faults.

- Same page may be brought into memory several times.

Page Replacement

- Page replacement is basic to demand paging

- Page replacement completes separation between logical memory and physical memory:

- Large virtual memory can be provided on a smaller physical memory.

- Use **modify bit (dirty bit)** to reduce overhead of page transfers – only modified pages are written to disk.

Basic Page Replacement

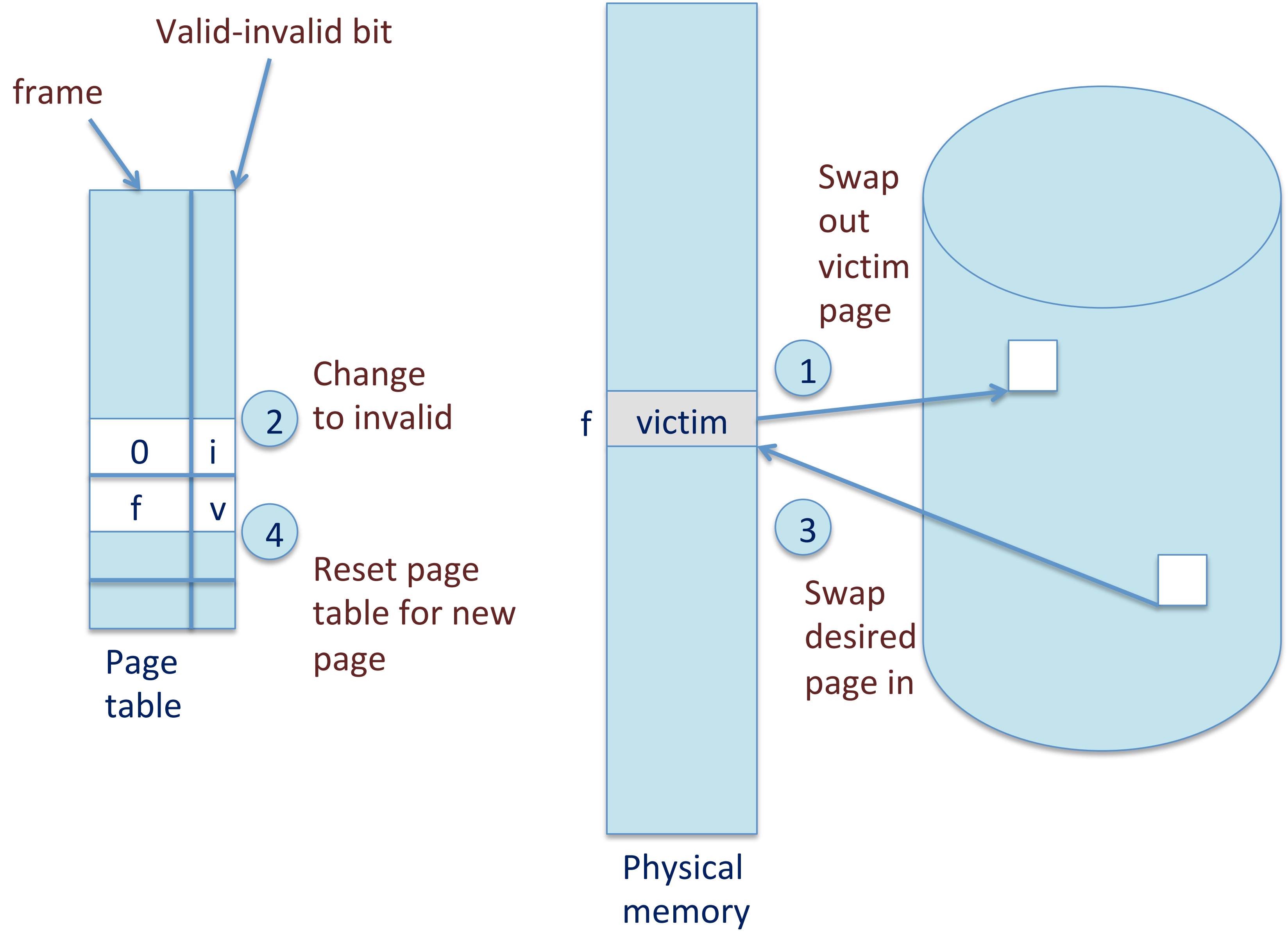
1. Find the location of the desired page on disk.

2. Find a free frame:
   - If there is a free frame, use it.
   - If there is no free frame, use a page replacement algorithm to select a victim frame.

3. Read the desired page into the (newly) free frame. Update the page and frame tables.

4. Restart the process.

Page Replacement



Acknowledgments

- These slides are adapted from
  - Öznur Özkasap (Koç University)
  - Operating System and Concepts (9th edition) Wiley