# Memory Management

# What is memory?

- A large array of words or bytes each with its own address.
- Central to the operation of a modern computer system.
- Instruction cycle –
  - ✓ Fetching the instruction from memory.
  - ✓ Decoding of the instruction.
  - ✓ Fetching of operands from memory.
  - ✓ Execution of operation.
  - ✓ Storing results back in memory.

# Address Binding

- A program resides on a disk as a binary executable file.
- The collection of processes on the disk that are waiting to be brought into memory for execution forms the input queue.
- Load the process.
- Access instructions and data from memory.
- After termination, declare that memory space available.

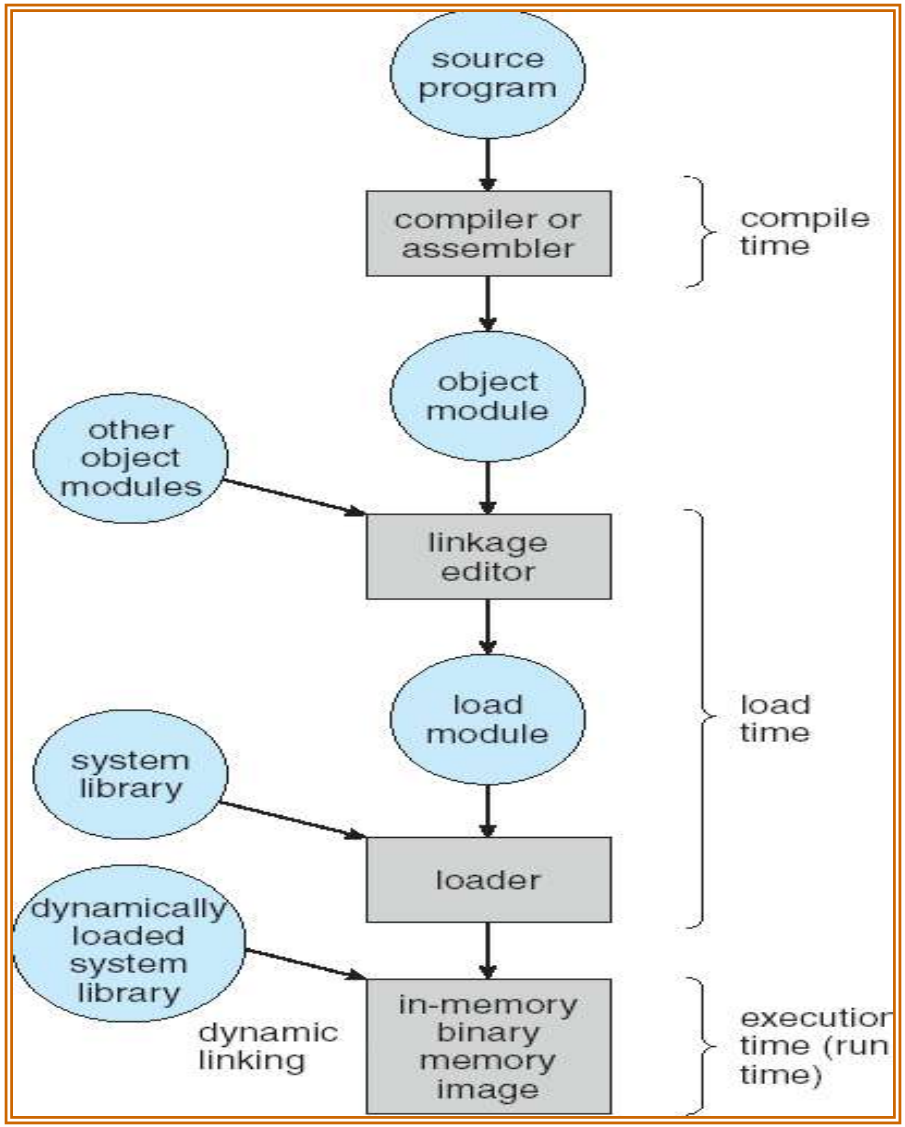
- Most systems allow a user process to reside in any part of the physical memory.
- The address space of the computer starts at 00000, the first address of the user process does not need to be 00000.
- Multiple steps before a user program is executed.
- Different ways of address representation.

- Address binding is a mapping from one address space to another.
- The binding of instructions and data to memory addresses can be done at –
  - ✓ Compile time – If it is known at compile time where the process will reside in memory, then absolute code can be generated. The code needs to be recompiled if the starting location changes.
  - ✓ Load time - Must generate relocatable code if memory location is not known at compile time. In this case final binding is delayed until load time.

- ✓ Execution time – If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time.

  Needs special hardware.

# Multistep processing of a user program



# Dynamic loading

- A routine is not loaded until it is called.
- All routines are kept on disk in a relocatable load format.
- Relocatable linking loader – loads the desired routine into memory, updation of program's address tables.
- Benefits – unused routine never loaded leading to better memory-space utilization.

  doesn't require special support from the operating system.

# Resident monitor

- Predecessor to the operating system.
- A piece of system software (program) to transfer control automatically from one job to the next.
- Always in memory.
- When the computer was turned on, the resident monitor was invoked which transferred control to a program. When the program terminated, control given back to resident monitor which would then go on to the next program.

# Overlays

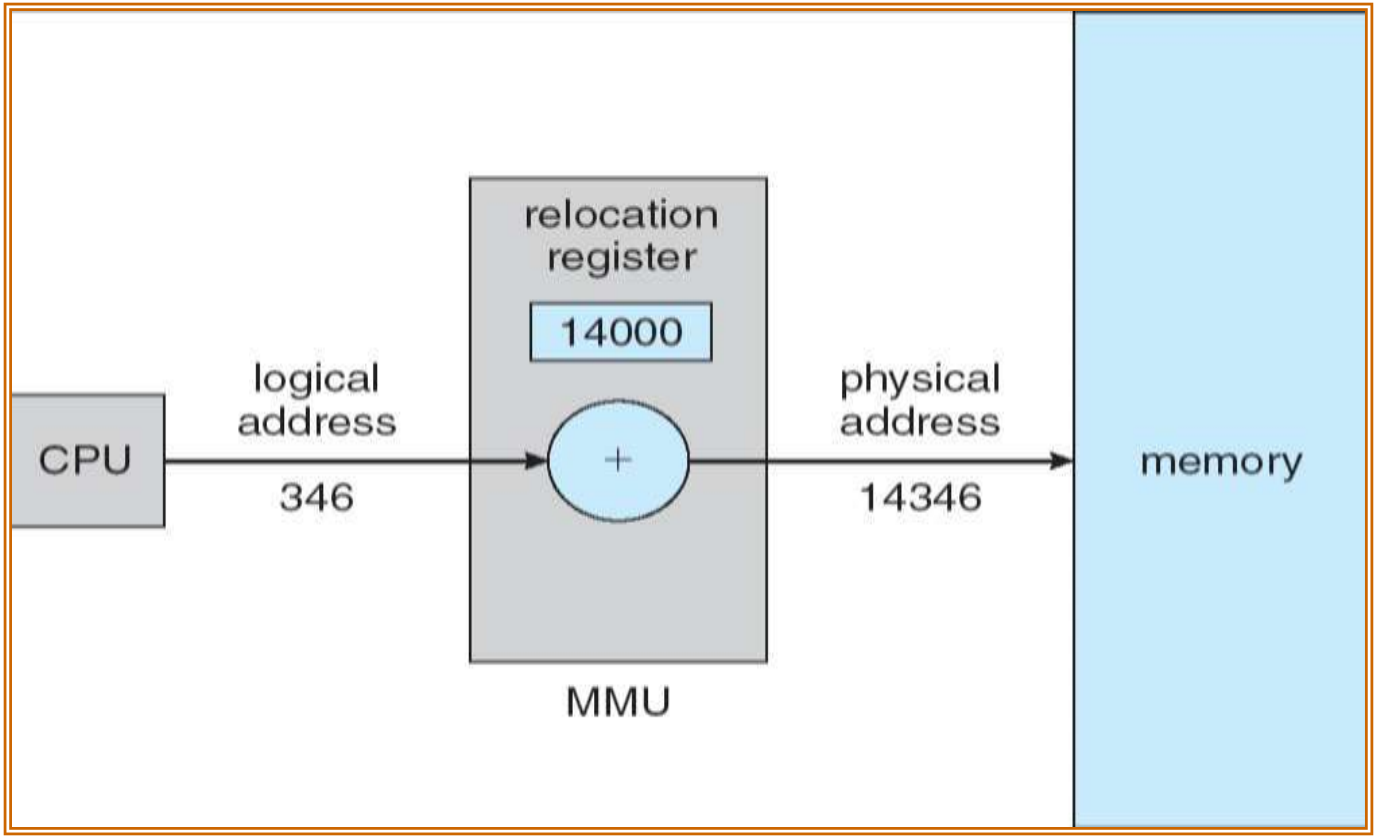
- A technique that is used so that a process can be larger than the amount of memory allocated to it.
- IDEA – to keep in memory only those instructions and data that are needed at any given time. When other instructions are needed, they are loaded into space that was occupied previously by instructions that are no longer needed.

- No special support needed from the operating system.
- Overlay structure needs to be designed and programmed properly.

# Address spaces

- *Logical / virtual address* – an address generated by the CPU.
- *Physical address* – an address seen by the memory unit.
- The set of all logical addresses generated by a program is known as a logical address space.
- The set of all physical addresses corresponding to these logical addresses is known as a physical address space.

- *Memory Management Unit (MMU)* – a hardware device that does the run-time mapping from virtual to physical addresses.

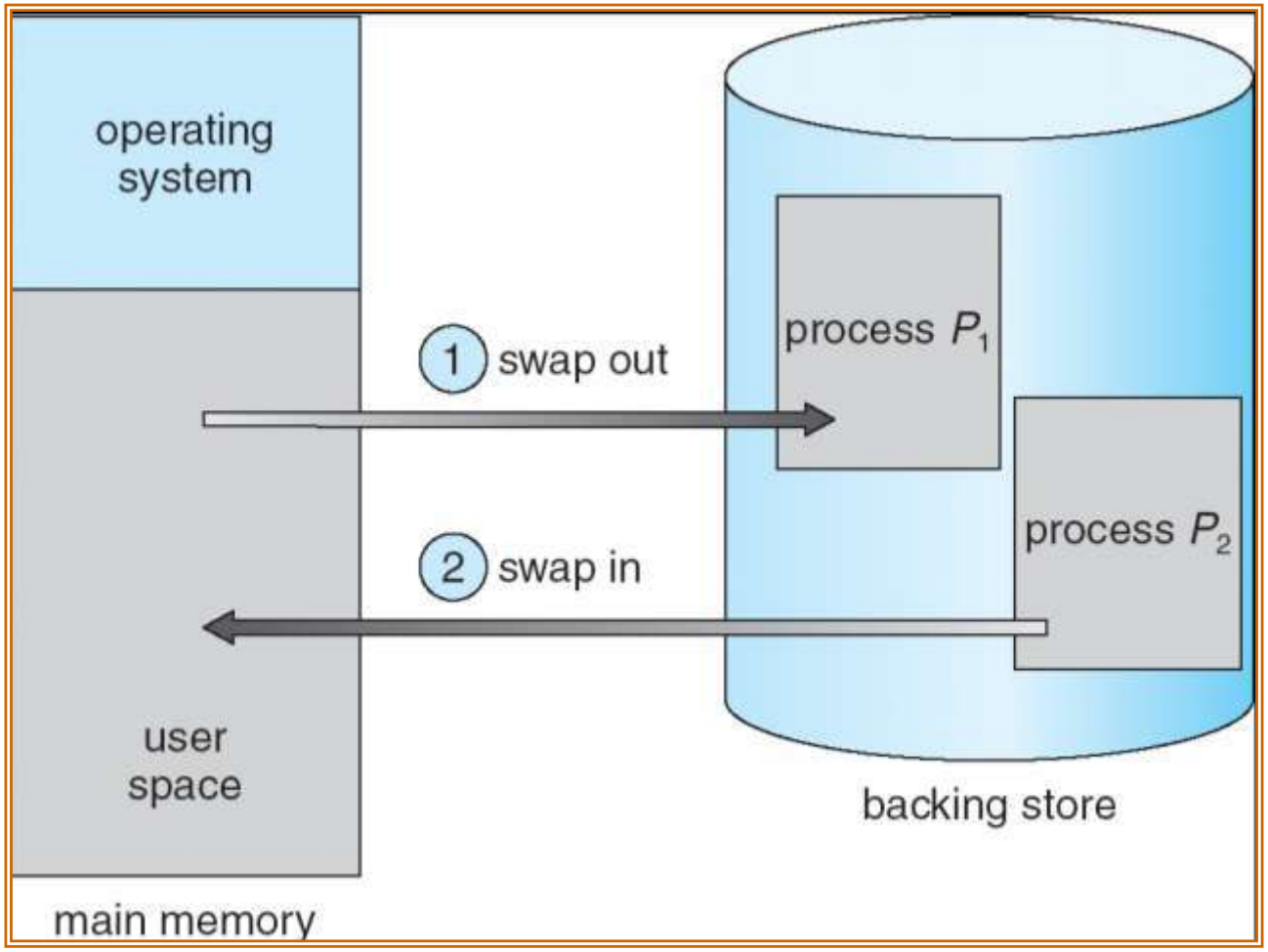


- The user program never sees the real physical addresses.
- It deals with logical addresses.

# Swapping

- a mechanism in which a process can be taken temporarily out of main memory to a backing store(a fast disk) and then brought back into memory for continued execution.
- Roll out, roll in – a swapping variant used for priority-based scheduling algorithms; a lower-priority process is swapped out by the memory manager so that a higher-priority process can be loaded and executed.

# Schematic View of Swapping



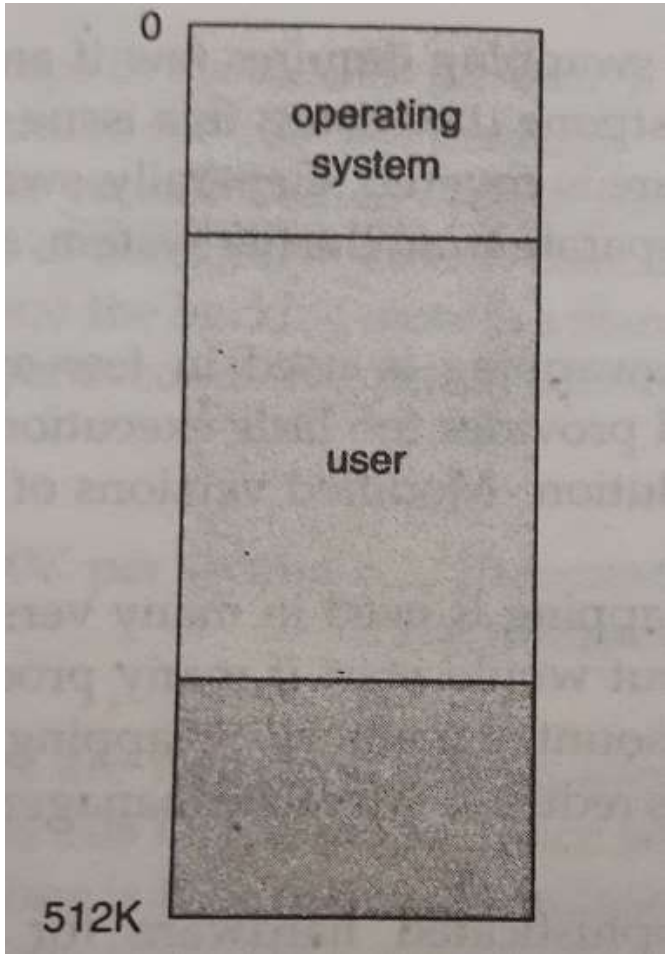
# Swapping contd..

- For efficient CPU utilization, the execution time for each process should be substantially greater than the swap time.
- The major part of the swap time is transfer time. The total transfer time is directly proportional to the amount of memory swapped.

# Contiguous Memory Allocation

- The main memory needs to accommodate both the operating system and the various user processes.
- Several user processes to reside in memory at the same time.
- Each process is contained in a single contiguous section of memory.

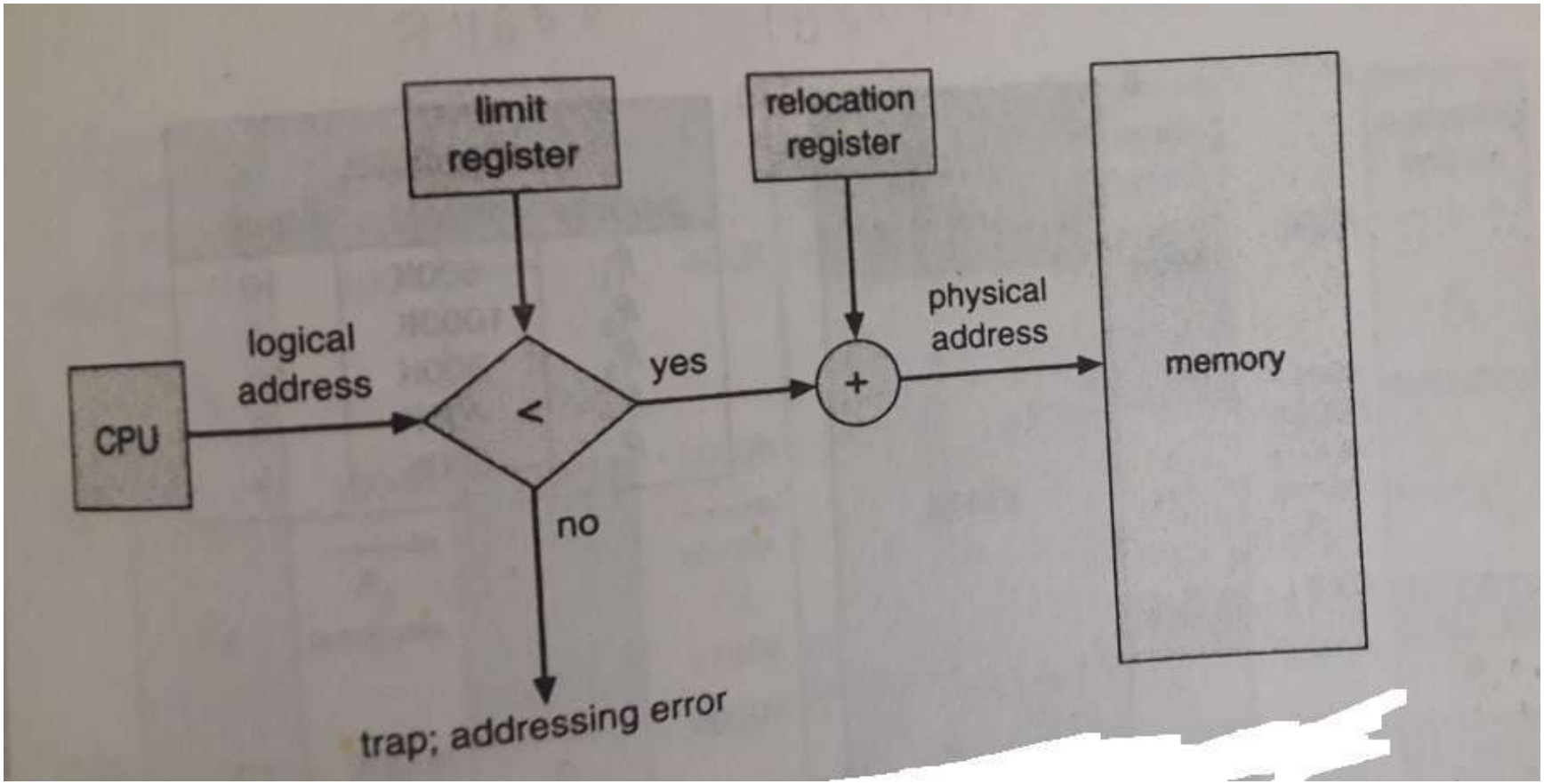
# Memory partition



# Memory protection

- Need to protect the operating-system code and data from changes by the user processes.
- User processes also need to be protected from one another.
- Using relocation register and limit register.
- Relocation register – holds the value of the smallest physical address.
- Limit register – contains the range of logical addresses.

# Hardware support for relocation and limit registers



# Memory Allocation

- Division of  memory into several fixed-sized partitions (multiple-partition method).
- Each partition may contain exactly one process.
- The degree of multiprogramming is bound by the number of partitions.
- No longer in use.

# Generalization of the fixed-partition scheme

- The operating system keeps a table indicating which parts of memory are available and which are occupied.
- Initially, memory is considered as one large block of available memory, a hole.
- A process that arrives is allocated only as much memory as is needed

# Dynamic storage-allocation problem

- How to satisfy a request of size n from a list of free holes?
- Solutions –
  - ✓ First-fit : Allocate the first hole that is big enough.
  - ✓ Best-fit : Allocate the smallest hole that is big enough.
  - ✓ Worst-fit : Allocate the largest hole.

# Fragmentation

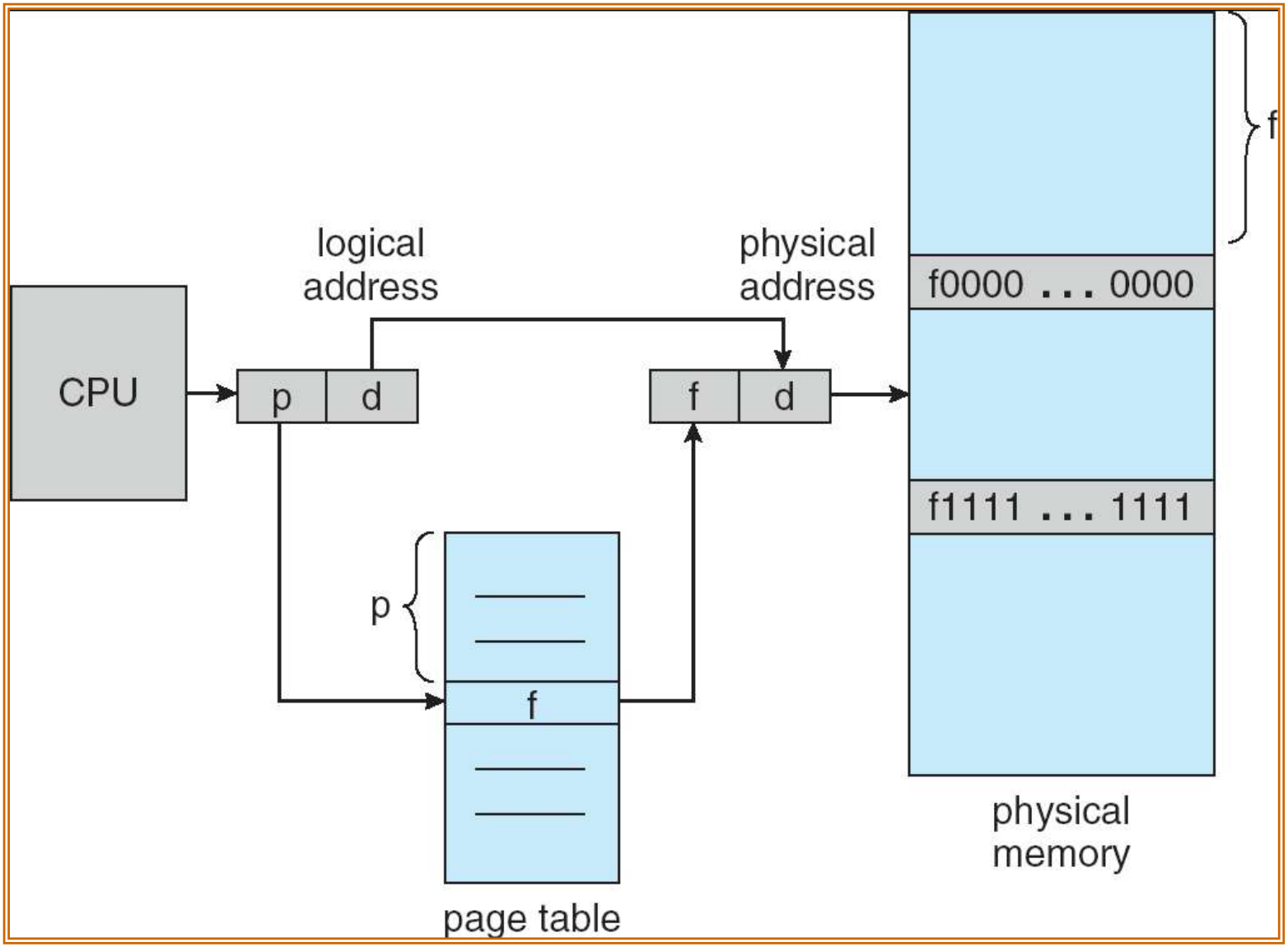
- External fragmentation – exists when enough total memory space exists to satisfy a request but it is not contiguous.
- Internal fragmentation – memory that is internal to a partition but is not being used.
- Solutions to external fragmentation –
  - ✓ Compaction – to shuffle the memory contents to place all free memory together in one large block.
  - ✓ to permit the logical address space of a process to be noncontiguous (paging).

# PAGING

- A memory management scheme that allows the logical address space of a process to be noncontiguous.
- Process is allocated physical memory wherever it is available.
- *Physical memory* broken into fixed-sized blocks called *frames*.
- *Logical memory* broken into blocks of the same size called *pages* (power of 2 varying between 512 bytes and 8192 bytes per page).

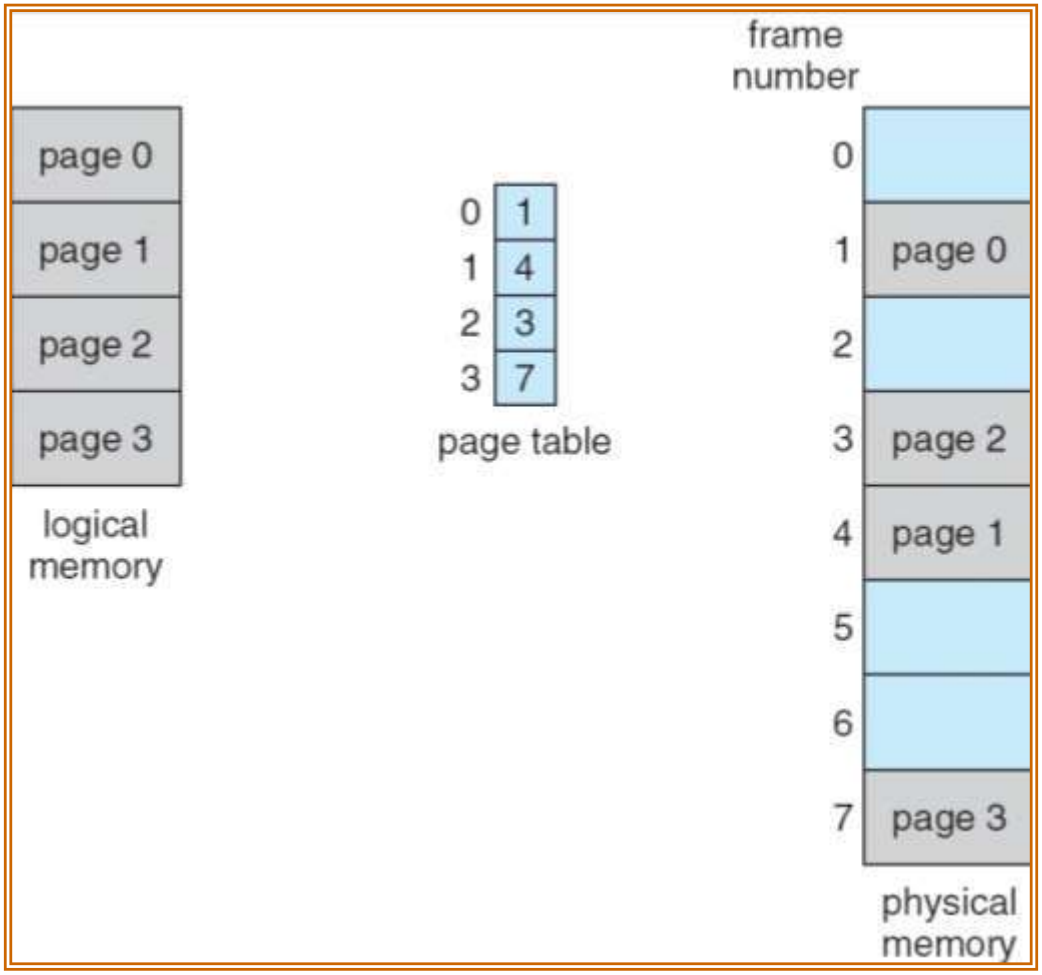
- Page size / frame size is defined by the hardware.
- To run a program of size n pages, need to find n free frames and load program.
- Set up a page table to translate logical to physical addresses.
- *Logical address divided into a page number and a page offset.*
- Page number is used as an index into a page table.
- The page table contains the base address of each page in physical memory.
- Page offset is the displacement within the page.

# Paging Hardware

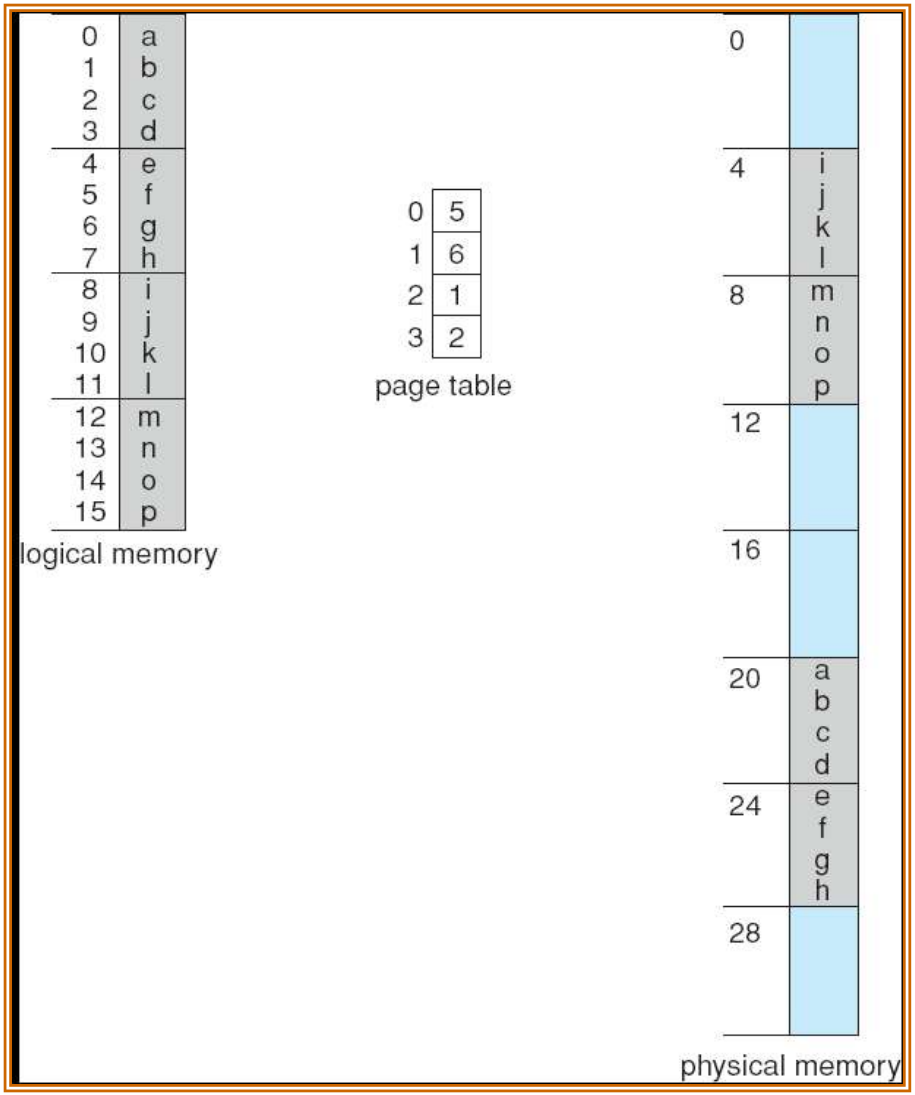


- If the size of logical address space is 2^m and a page size is 2^n addressing units (bytes or words), then the high-order m-n bits of a logical address designate the page number and the n low-order bits designate the page offset.
- Que – If the size of the logical address space is 16B, page size is 4B, main memory is 32B, how many frames would be there?

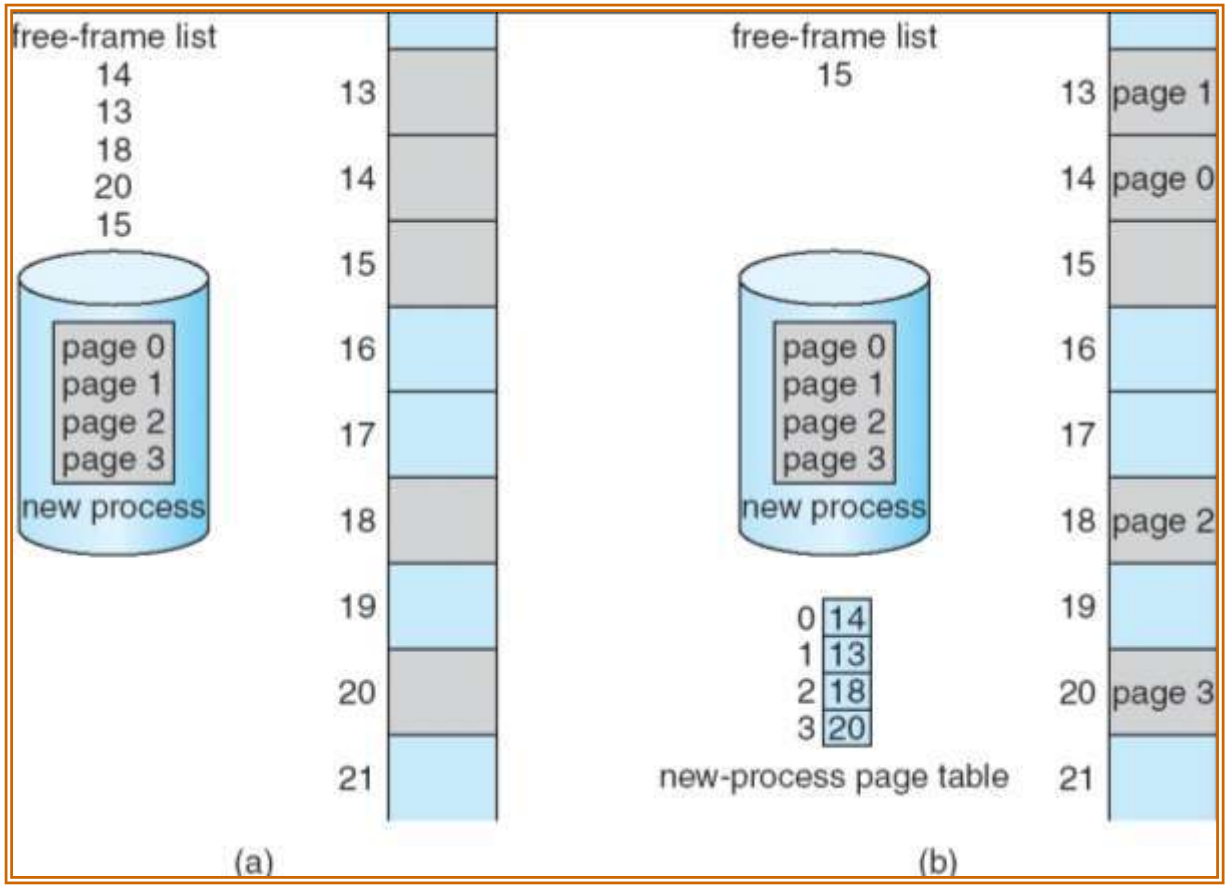
# Paging model of logical and physical memory



# Paging example for a 32-byte memory with 4-byte pages



- Internal fragmentation.
- Each page of the process needs one frame.



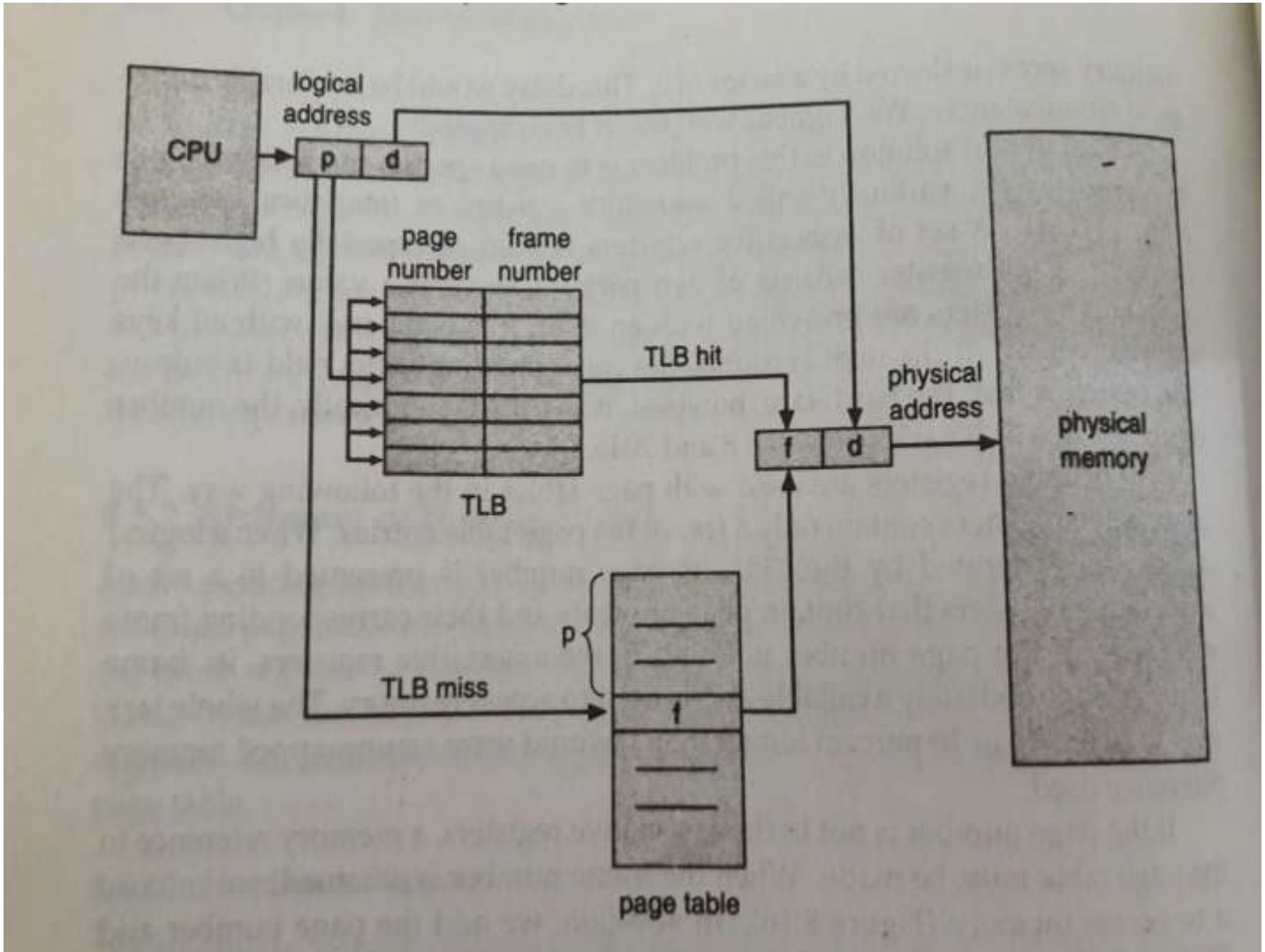
- Operating system needs to be aware of the allocation details of the physical memory.
- This information is kept in frame table.
- One entry for each physical page frame.

# Hardware implementation of page table

- Page table implemented as a set of dedicated registers.
- High-speed logic.
- Works well for small page tables.
- For large page tables, page table kept in memory and a page-table base register (PTBR) points to the page table.
- Problem – two memory accesses needed to access a byte.

- Solution is to use associative registers or translation look-aside buffers (TLBs).
- A special, small, fast-lookup hardware cache.
- Contain only few of the page-table entries.
- The percentage of times that a page number is found in the associative registers is called the hit ratio.
- Effective memory-access time = Hit time X Hit ratio + Miss time X Miss ratio

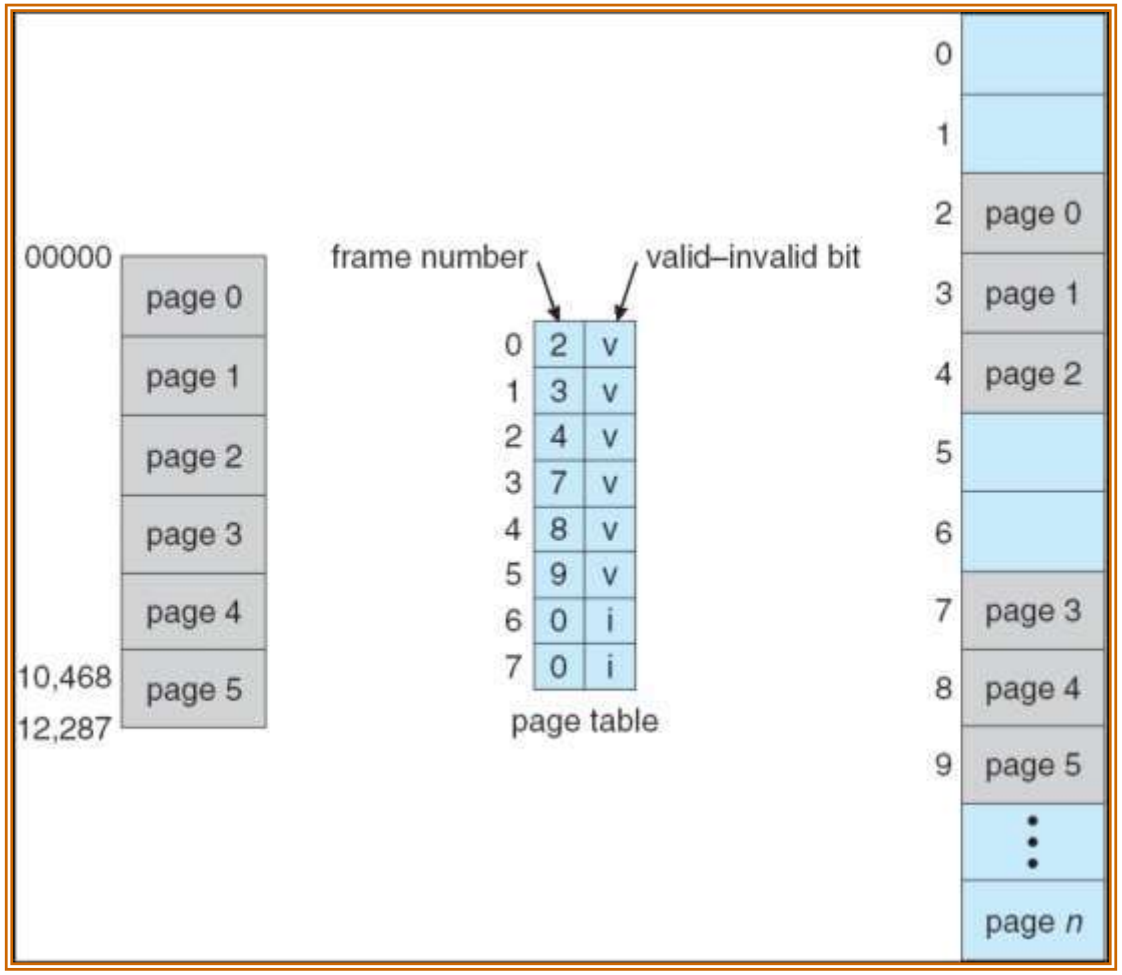
# Paging hardware with TLB



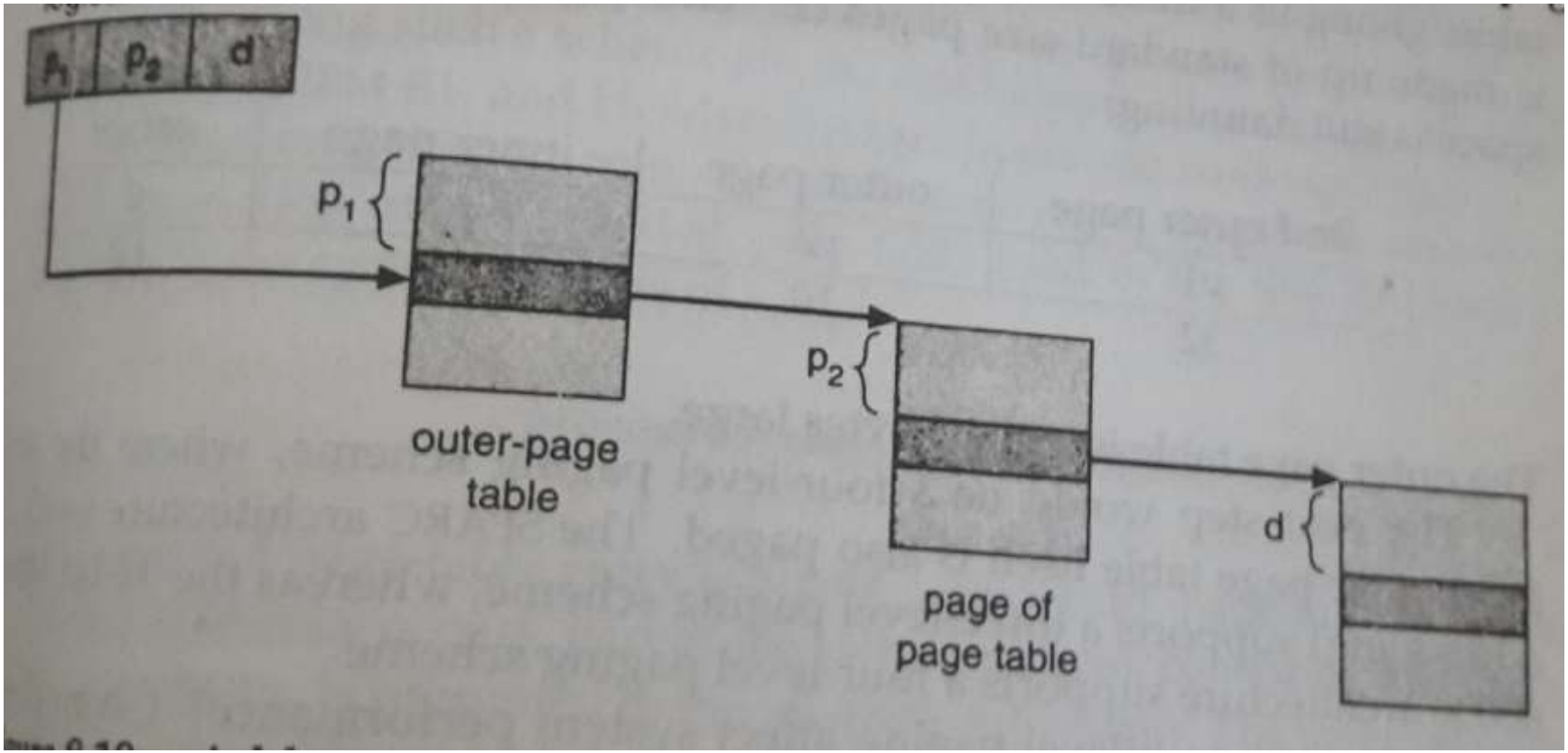
# Memory protection in a paged environment

- Protection bits associated with each frame.
- Bits kept in page table.
- Valid-invalid bit - when this bit is set to "valid," the associated page is in the process's logical address space and is thus a legal (or valid) page. When the bit is set to "invalid“, the page is not in the process's logical address space.
- Illegal addresses are trapped by use of the valid-invalid bit.

# Valid (v) or Invalid (i) Bit in a Page Table



# Multilevel Paging (Two level)



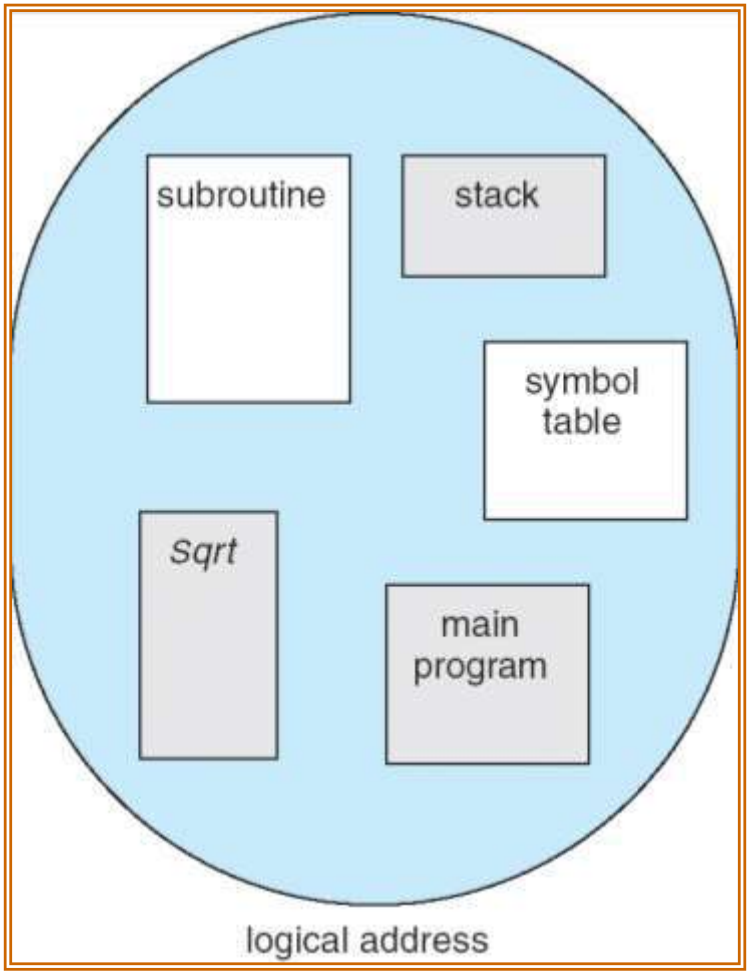
# Paging contd…

- Clear separation between the user's view of memory and the actual physical memory.
- Difference reconciled by the address translation hardware.
- Logical addresses translated into physical addresses.
- Mapping hidden from the user.
- Controlled by the operating system.

# SEGMENATION

- A memory-management scheme that supports the user view of memory.
- Memory is  a collection of variable-sized segments with no necessary ordering among segments.
- Logical address space is  a collection of segments.
- Each segment has a name and a length.
- The user specifies each address by a segment name and an offset.

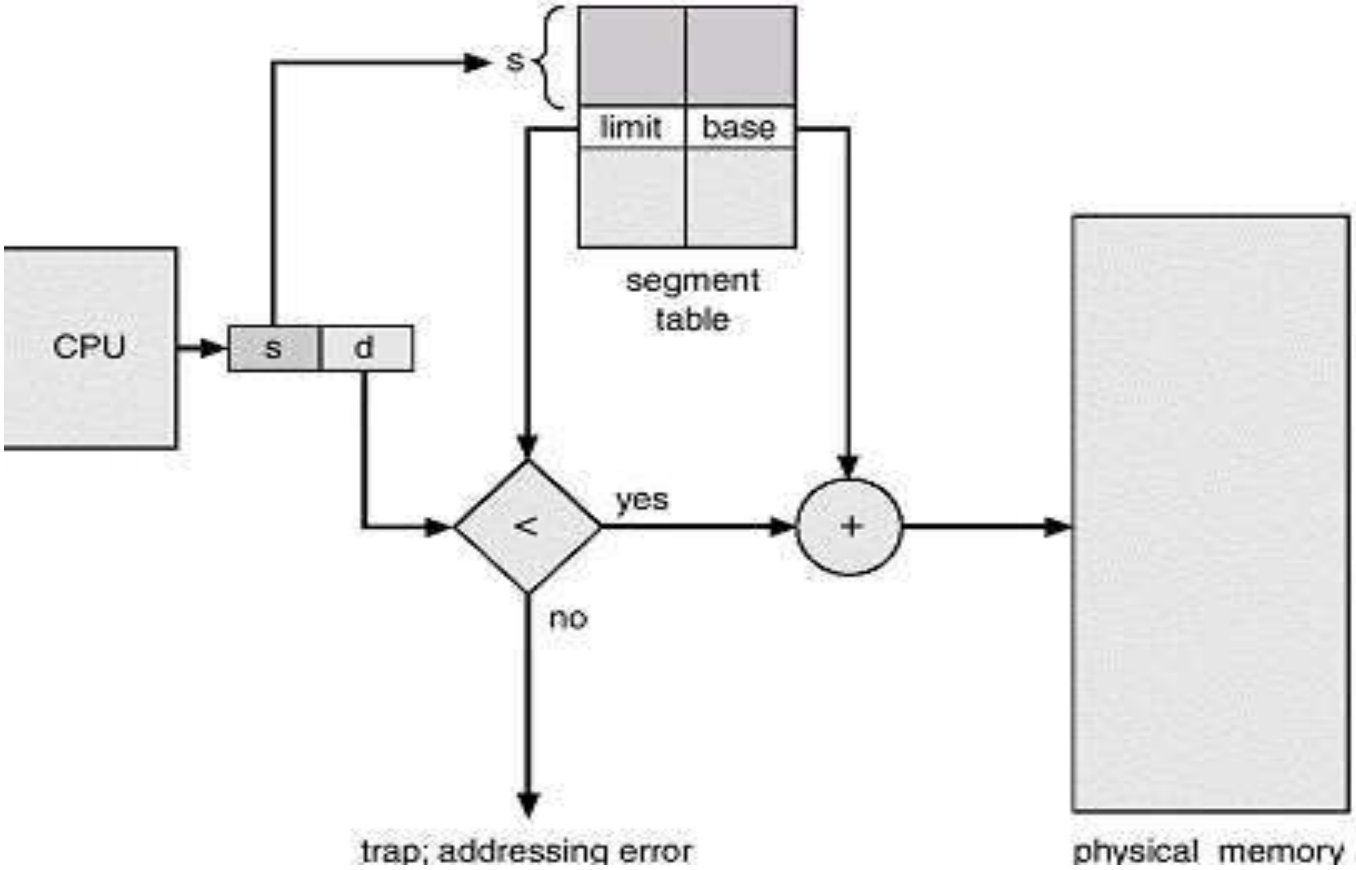
# User's view of a program



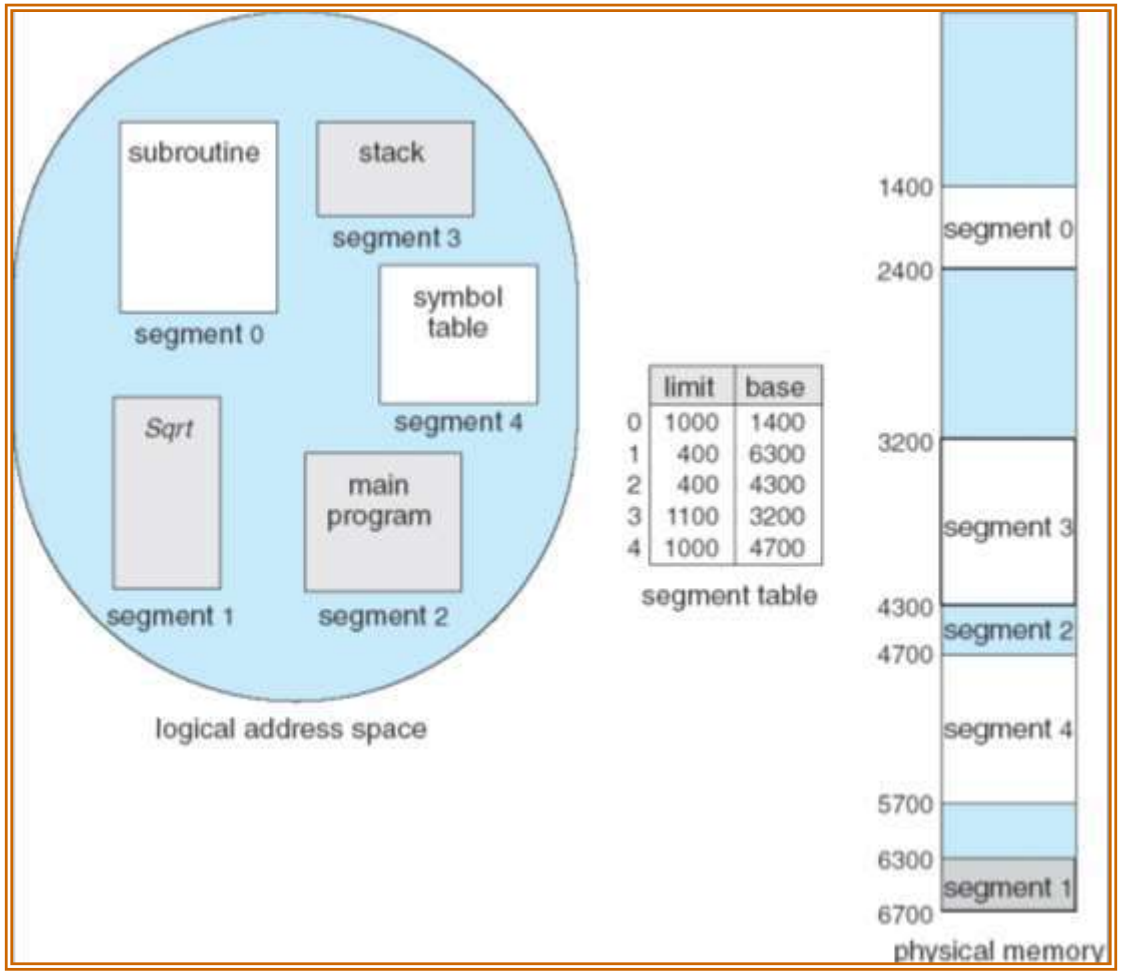
# Hardware

- Need to map two-dimensional user-defined addresses into one-dimensional physical addresses.
- Done by segment table.
- *Segment base* contains the beginning address of the segment in physical memory.
- *Segment limit* specifies the length of the segment.

# Segmentation hardware



# Example of segmentation

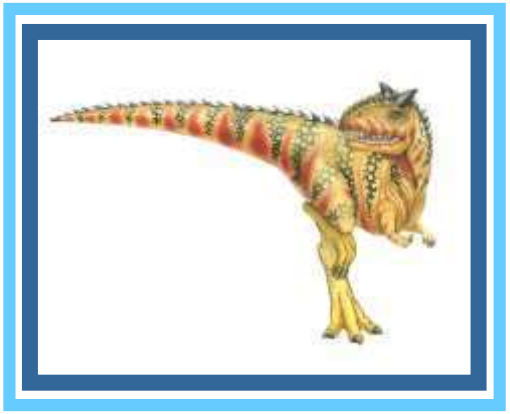


# Implementation of Segment Tables

- The segment table can be put either in fast registers or in memory.
- A segment-table base register (STBR) points to the segment table.
- A segment-table length register (STLR) is used to specify the number of segments used by a program.
- A segment number s is legal if s<STLR.
- Memory address of the segment-table entry

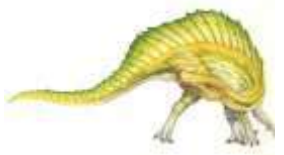
  = STBR + s.

# Chapter: Virtual Memory



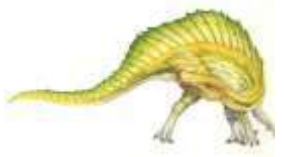
# Introduction

- Code needs to be in memory to execute, but entire program rarely used
  - Error code, unusual routines, large data structures
- Entire program code not needed at same time
- Consider ability to execute partially-loaded program
  - Program no longer constrained by limits of physical memory
  - Each program takes less memory while running -> more programs run at the same time
    - Increased CPU utilization and throughput with no increase in response time or turnaround time
  - Less I/O needed to load or swap programs into memory -> each user program runs faster



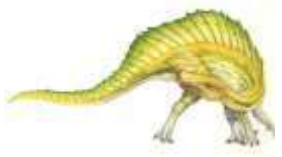
# Introduction

- **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
  - More programs running concurrently
  - Less I/O needed to load or swap processes

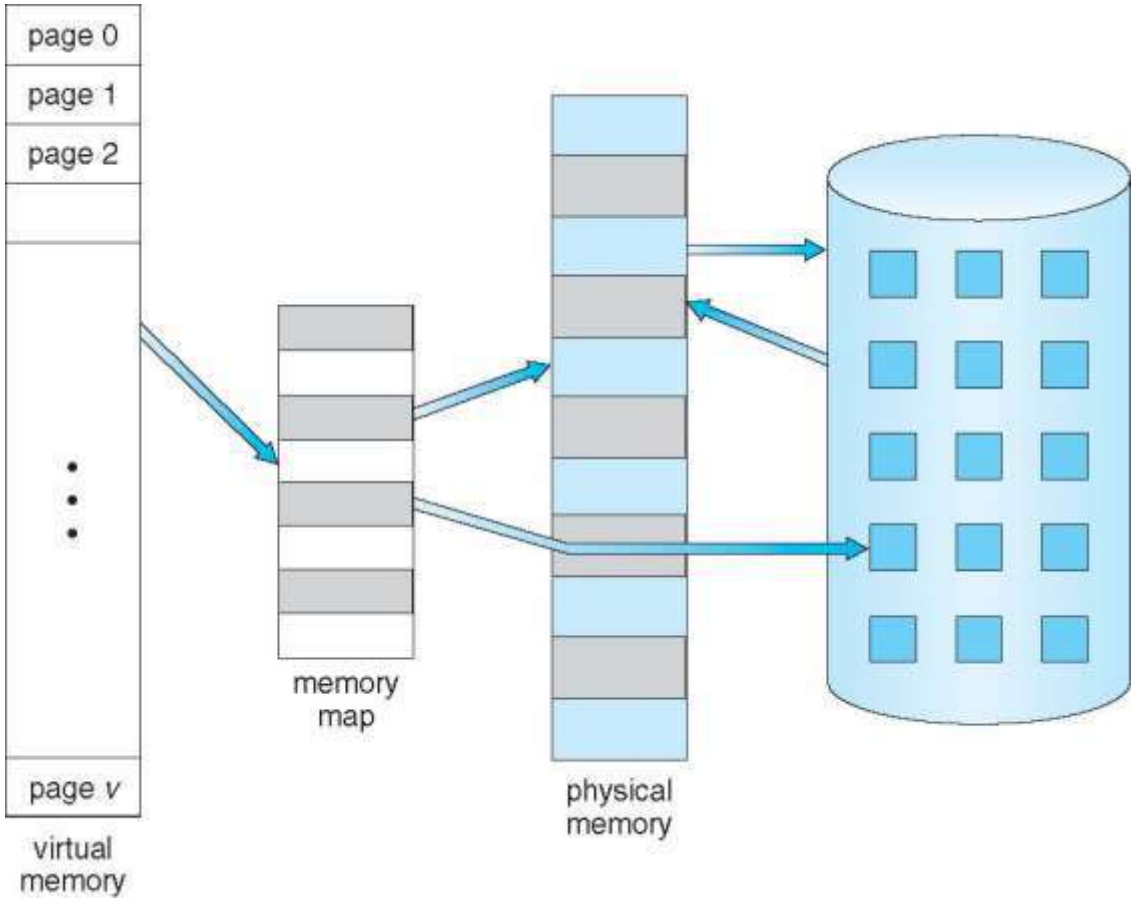


# Introduction

- **Virtual address space** – logical view of how process is stored in memory
  - Usually start at address 0, contiguous addresses until end of space
  - Meanwhile, physical memory organized in page frames
  - MMU must map logical to physical
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation



# Virtual Memory That is Larger Than Physical Memory

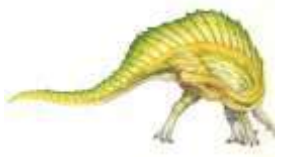


# Demand Paging

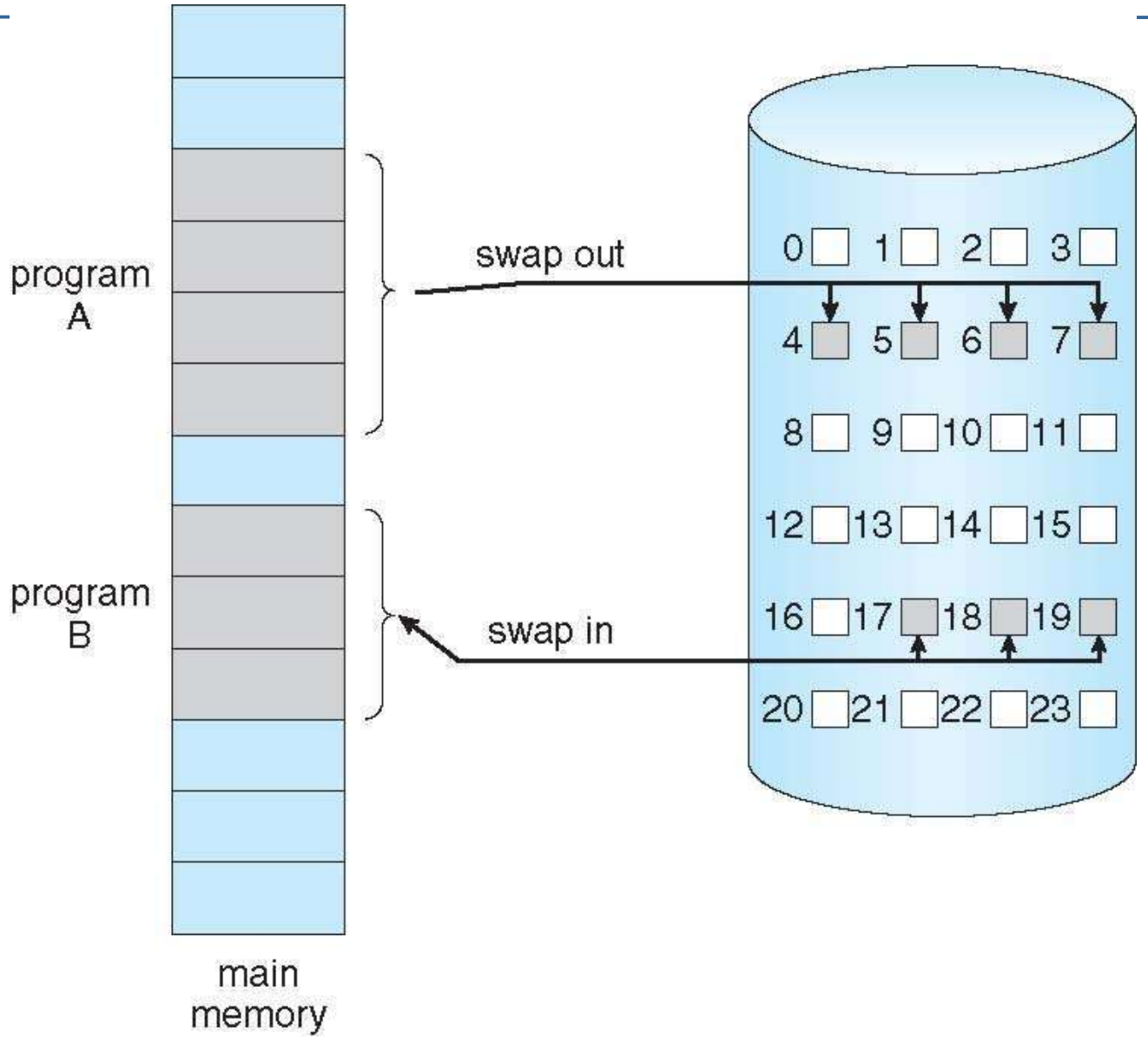
Could bring entire process into memory at load time Or bring a page into memory only when it is needed Less I/O needed, no unnecessary I/O

**Lazy swapper** – never swaps a page into memory unless page will be needed

Swapper that deals with pages is a **pager**



# Demand Paging

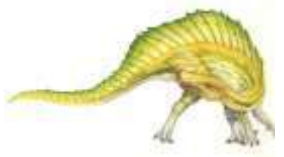


# Page Fault

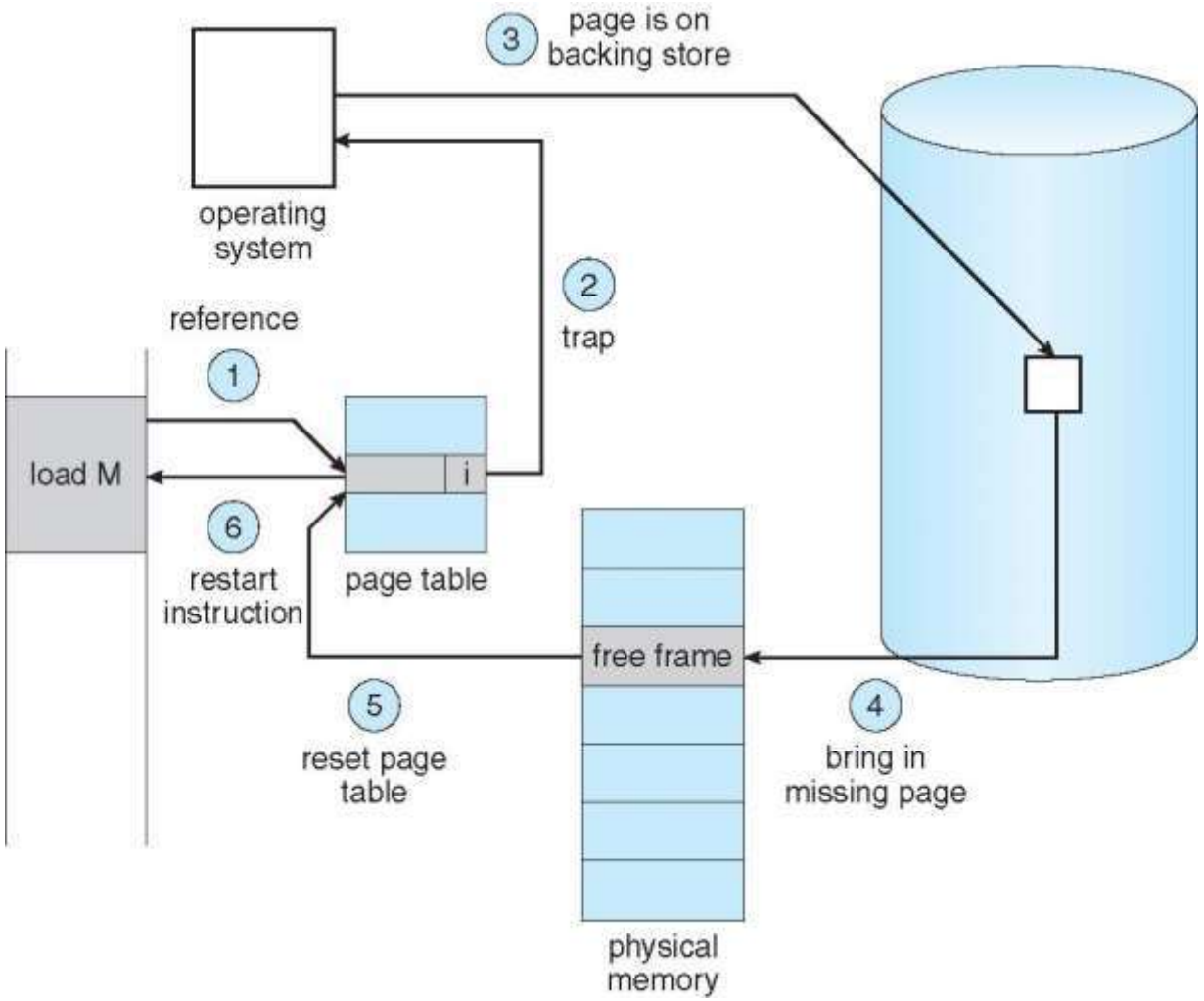
- If there is a reference to a page, first reference to that page will trap to operating system:

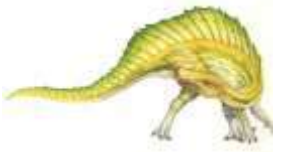
  **page fault**

1. Operating system looks at another table to decide:
   - Invalid reference ⇒ abort
   - Just not in memory
2. Find free frame
3. Swap page into frame via scheduled disk operation
4. Reset tables to indicate page now in memory
   Set validation bit = **v**
5. Restart the instruction that caused the page fault



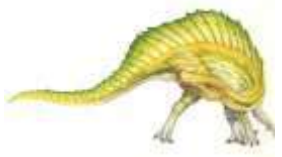
# Steps in Handling a Page Fault



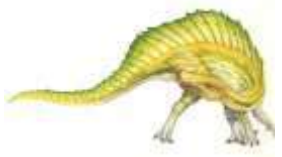
# Aspects of Demand Paging

- Start process with *no* pages in memory
  - OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault
  - And for every other process pages on first access
  - **Pure demand paging**
- Hardware support needed for demand paging
  - Page table with valid / invalid bit
  - Secondary memory (swap device with **swap space**)
  - Instruction restart



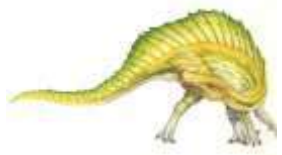
# What Happens if There is no Free Frame?

- Used up by process pages
- Also in demand from the kernel, I/O buffers, etc
- How much to allocate to each?
- Page replacement – find some page in memory, but not really in use, page it out
  - Algorithm – terminate? swap out? replace the page?
  - Performance – want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times

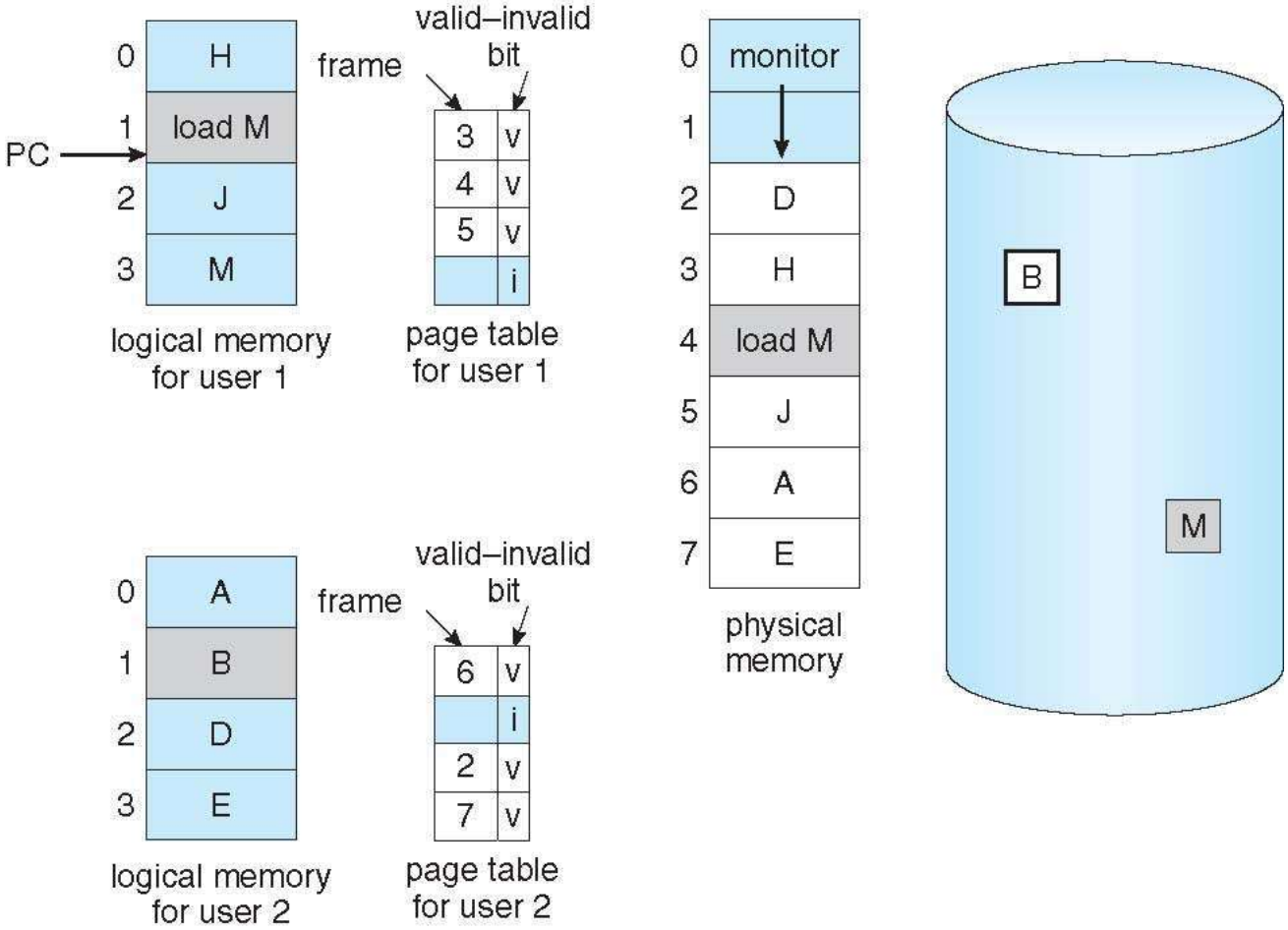


# Page Replacement

- Prevent **over-allocation** of memory by modifying page-fault service routine to include page replacement
- Use **modify** (**dirty**) **bit** to reduce overhead of page transfers – only modified pages are written to disk
- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory



# Need For Page Replacement



# Basic Page Replacement

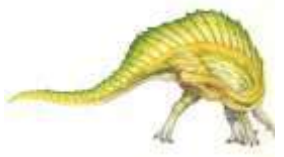
1. Find the location of the desired page on disk

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**
     - Write victim frame to disk if dirty

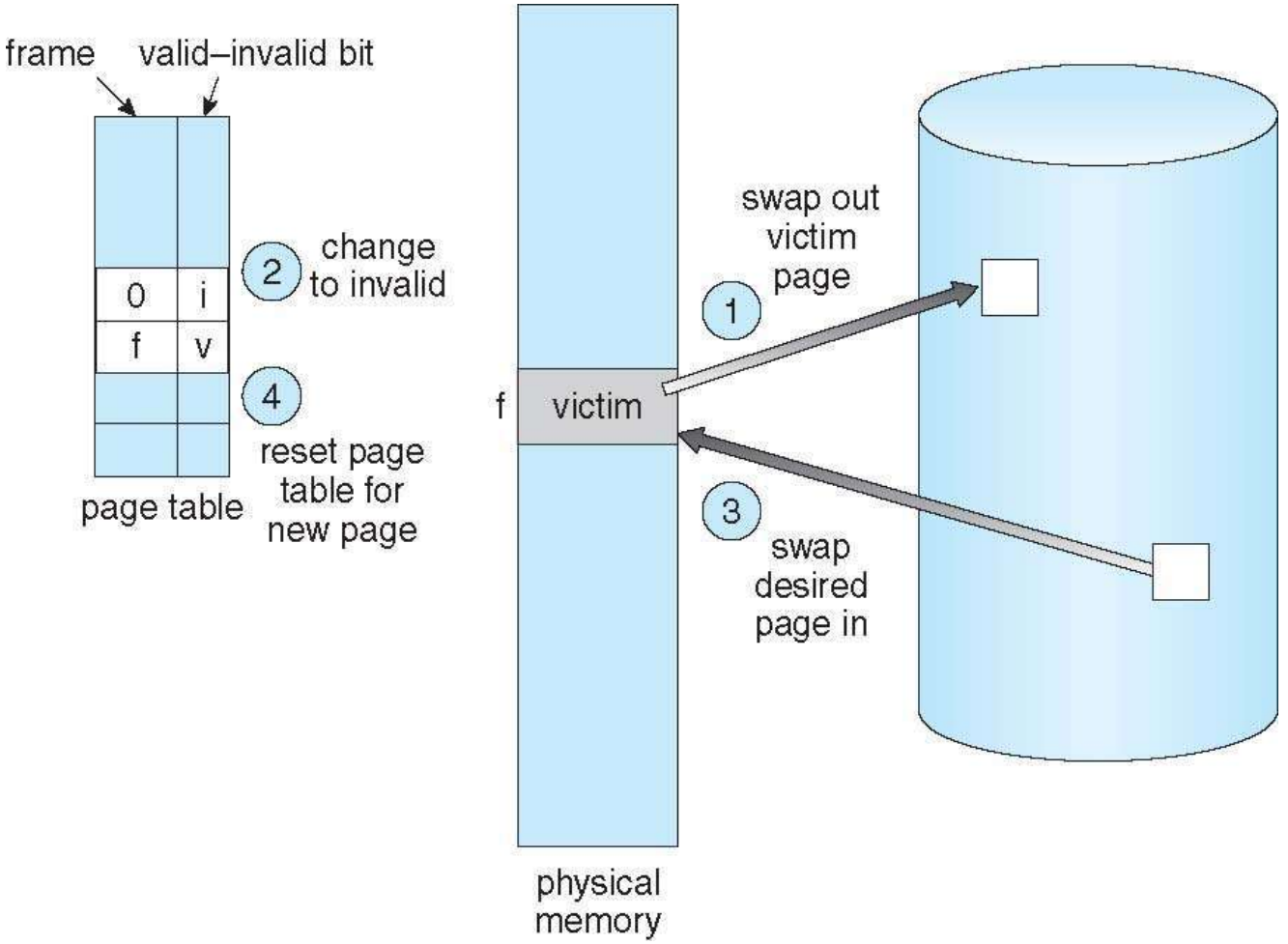
3. Bring the desired page into the (newly) free frame; update the page and frame tables

4. Continue the process by restarting the instruction that caused the trap

Note now potentially 2 page transfers for page fault – increasing EAT



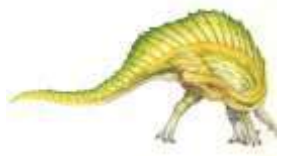
# Page Replacement



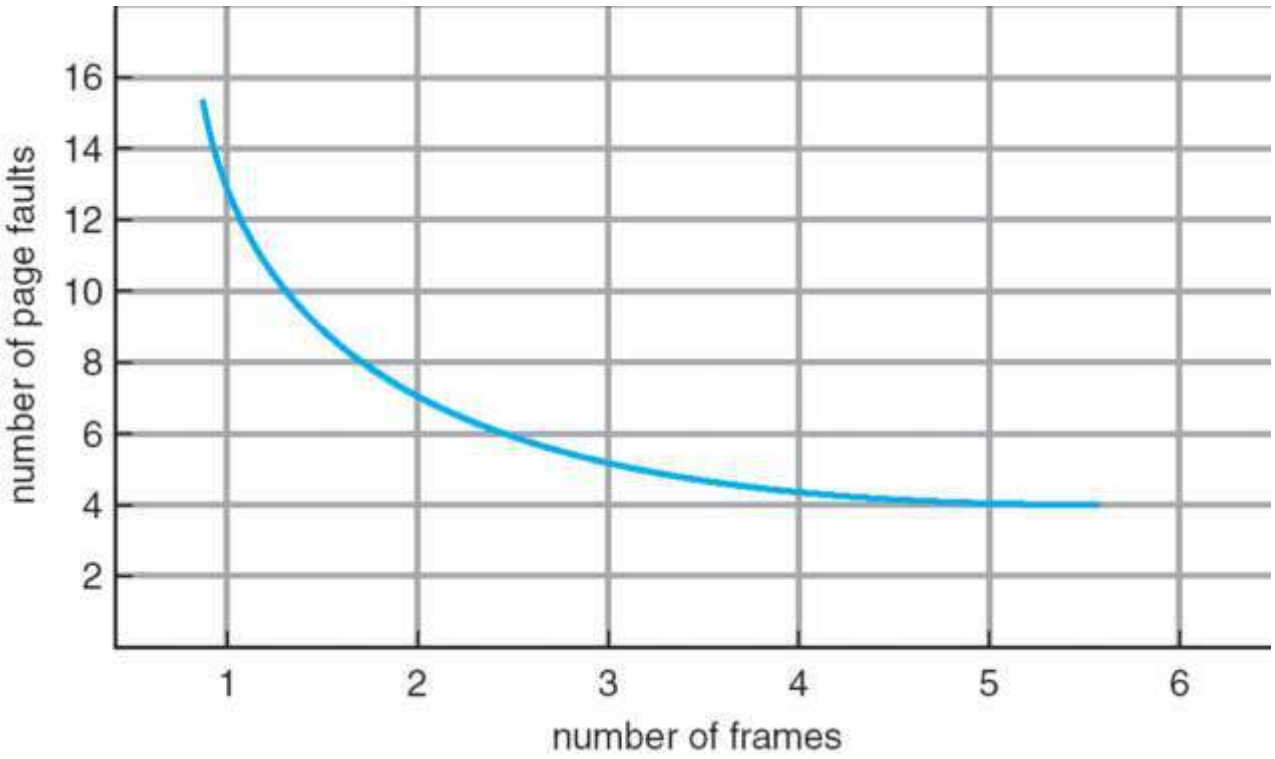
# Page and Frame Replacement Algorithms

- **Frame-allocation algorithm** determines
  - How many frames to give each process
  - Which frames to replace
- **Page-replacement algorithm**
  - Want lowest page-fault rate on both first access and re-access
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
  - String is just page numbers, not full addresses
  - Repeated access to the same page does not cause a page fault
  - Results depend on number of frames available
- In all our examples, the **reference string** of referenced page numbers is

**7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**

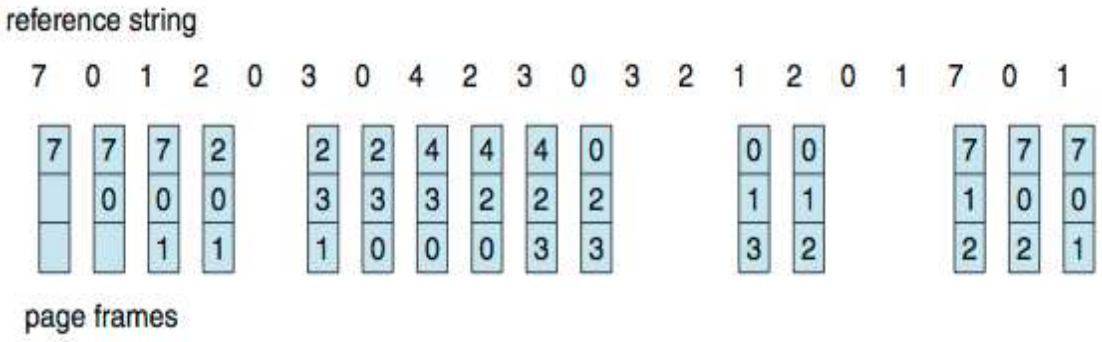


# Graph of Page Faults Versus The Number of Frames



# First-In-First-Out (FIFO) Algorithm

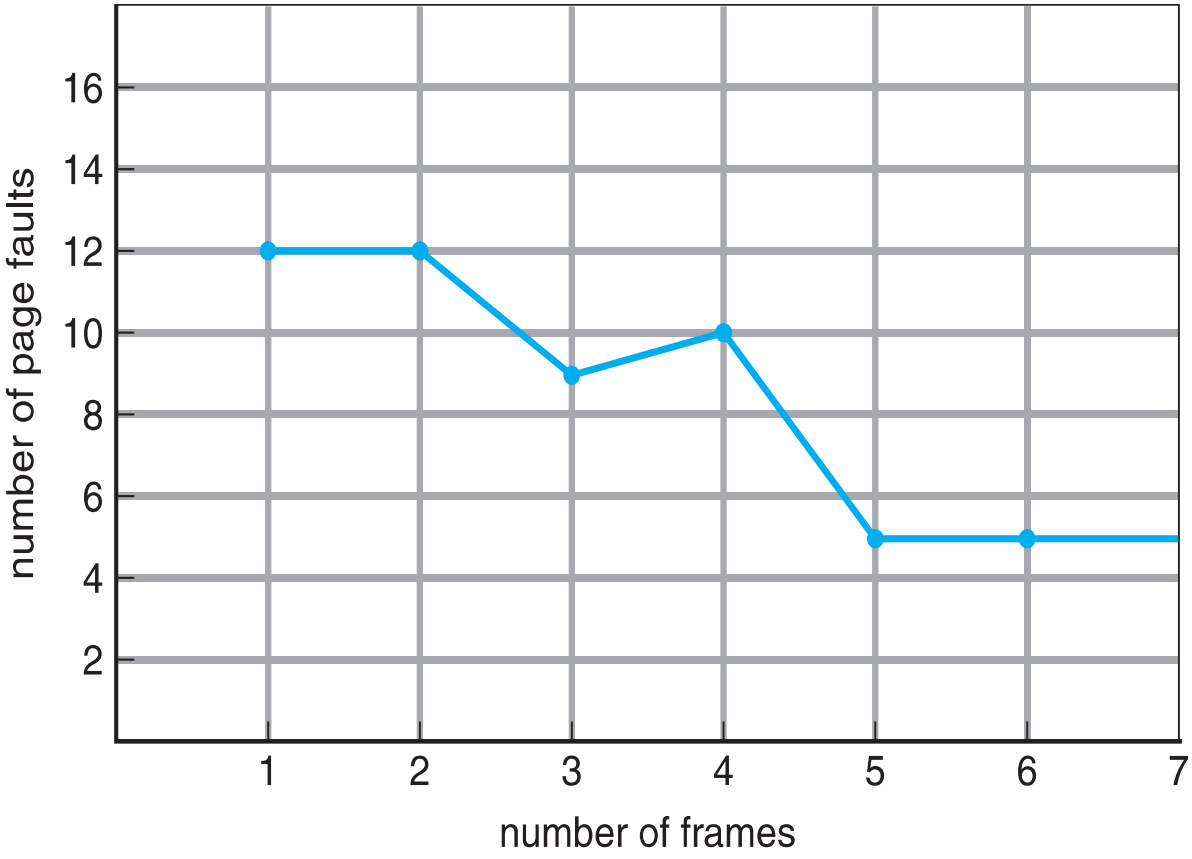
- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)




15 page faults

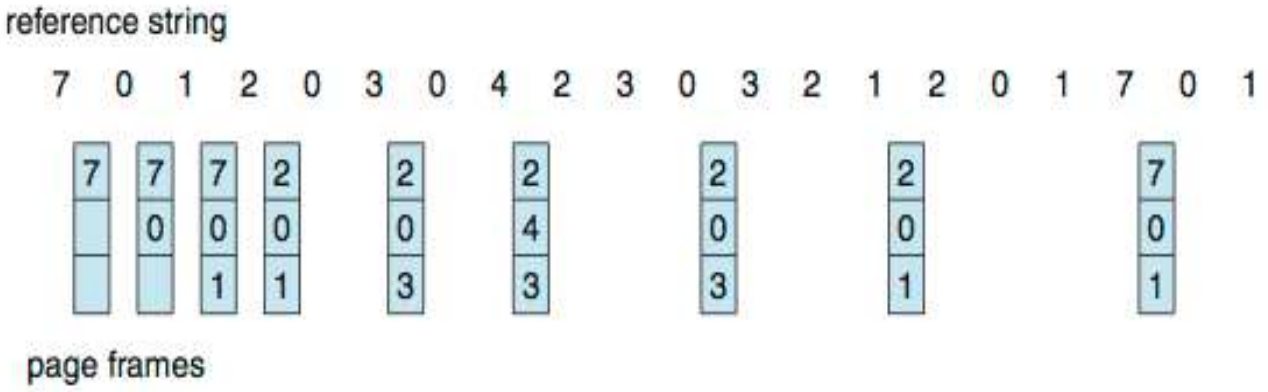
- Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5
  - Adding more frames can cause more page faults!
    - **Belady's Anomaly**
- How to track ages of pages?
  - Just use a FIFO queue

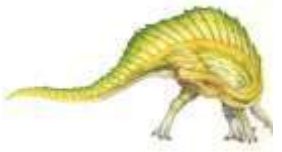
# FIFO Illustrating Belady's Anomaly



# Optimal Algorithm

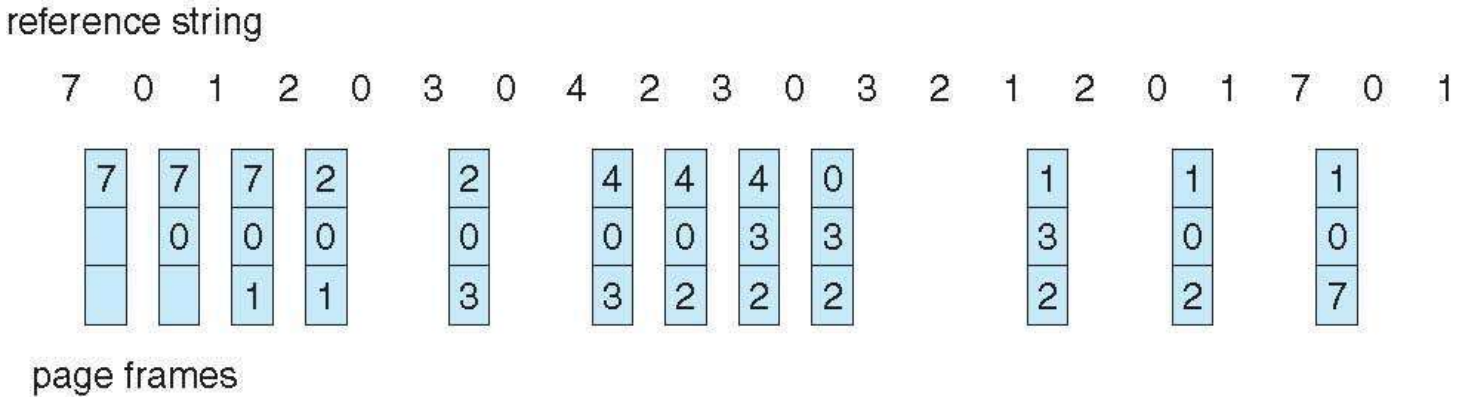
- Replace page that will not be used for longest period of time
  - 9 is optimal for the example
- How do you know this?
  - Can't read the future
- Used for measuring how well your algorithm performs



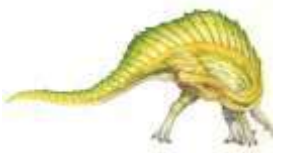



# Least Recently Used (LRU) Algorithm

- Use past knowledge rather than future
- Replace page that has not been used in the most amount of time
- Associate time of last use with each page

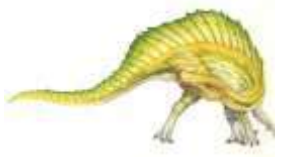



- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- But how to implement?

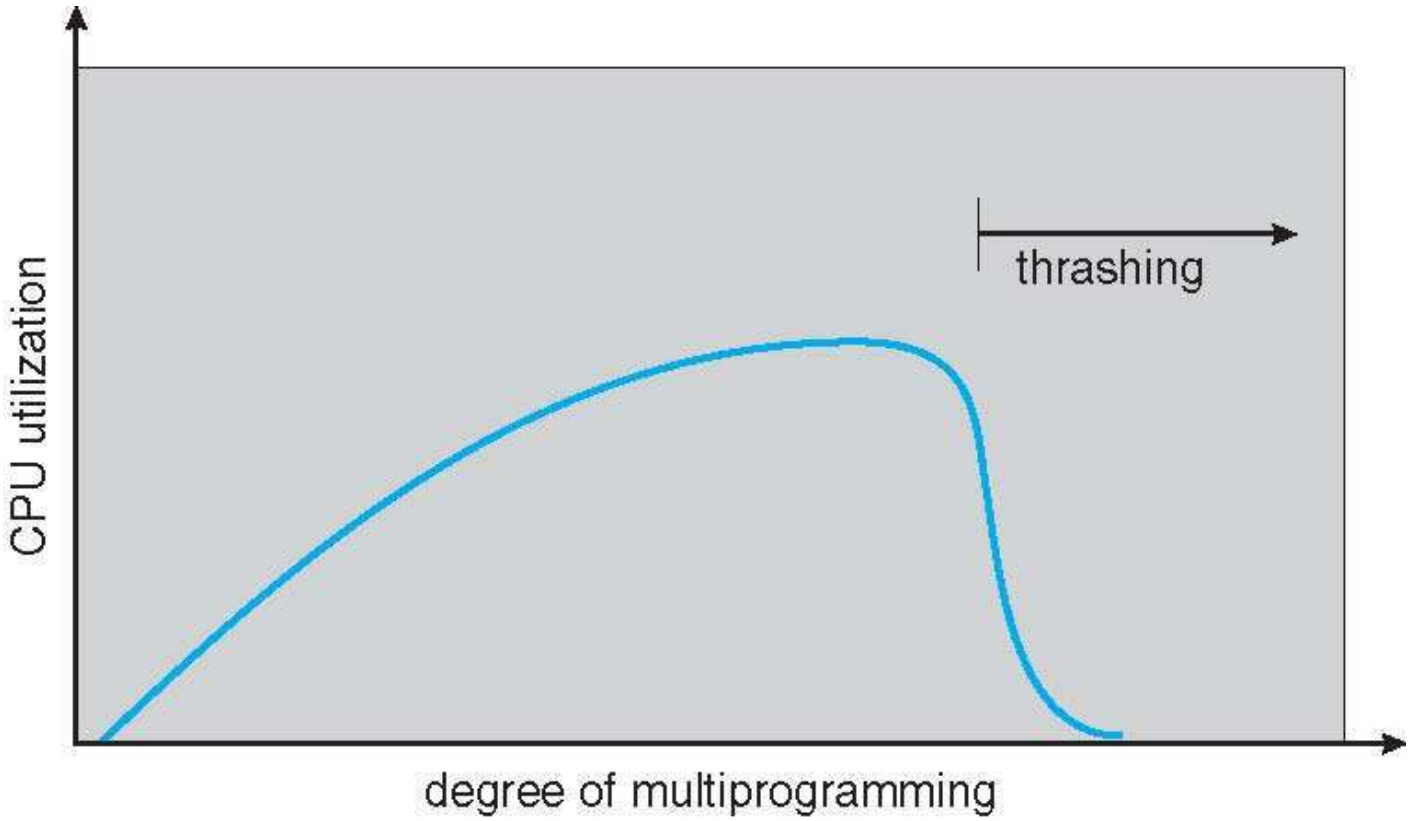


# Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high
  - Page fault to get page
  - Replace existing frame
  - But quickly need replaced frame back
  - This leads to:
    - Low CPU utilization
    - Operating system thinking that it needs to increase the degree of multiprogramming
    - Another process added to the system
- **Thrashing** ≡ a process is busy swapping pages in and out



# Thrashing (Cont.)



# End