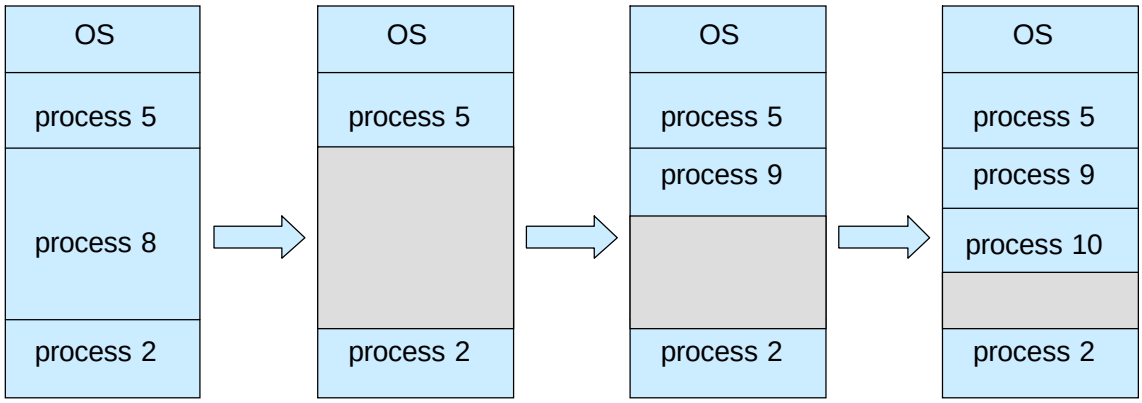
Lesson 6: Memory Management & Virtualization

Contents

- Dynamic allocation, Memory fragmentation
- Paging
- Address translation with paging
- Paging hardware
- Page table structures
- Segmentation
- Virtual memory background
- Demand Paging
- Page Replacement
- Allocation of Frames
- Thrashing
- Working sets

Holes in Memory Space

- Contiguous allocation produces holes
- Holes are produced also by other allocation strategies



Dynamic Storage-Allocation Problem

How to satisfy a request of size *n* from a list of free holes

- **First-fit**: Allocate the *first* hole that is big enough
  - Fastest method
- **Best-fit**: Allocate the *smallest* hole that is big enough;
  - Must search entire list, unless ordered by size. Produces the smallest leftover hole
  - Good storage utilization
- **Worst-fit**: Allocate the *largest* hole; must also search entire list. Produces the largest leftover hole.
  - Low storage fragmentation

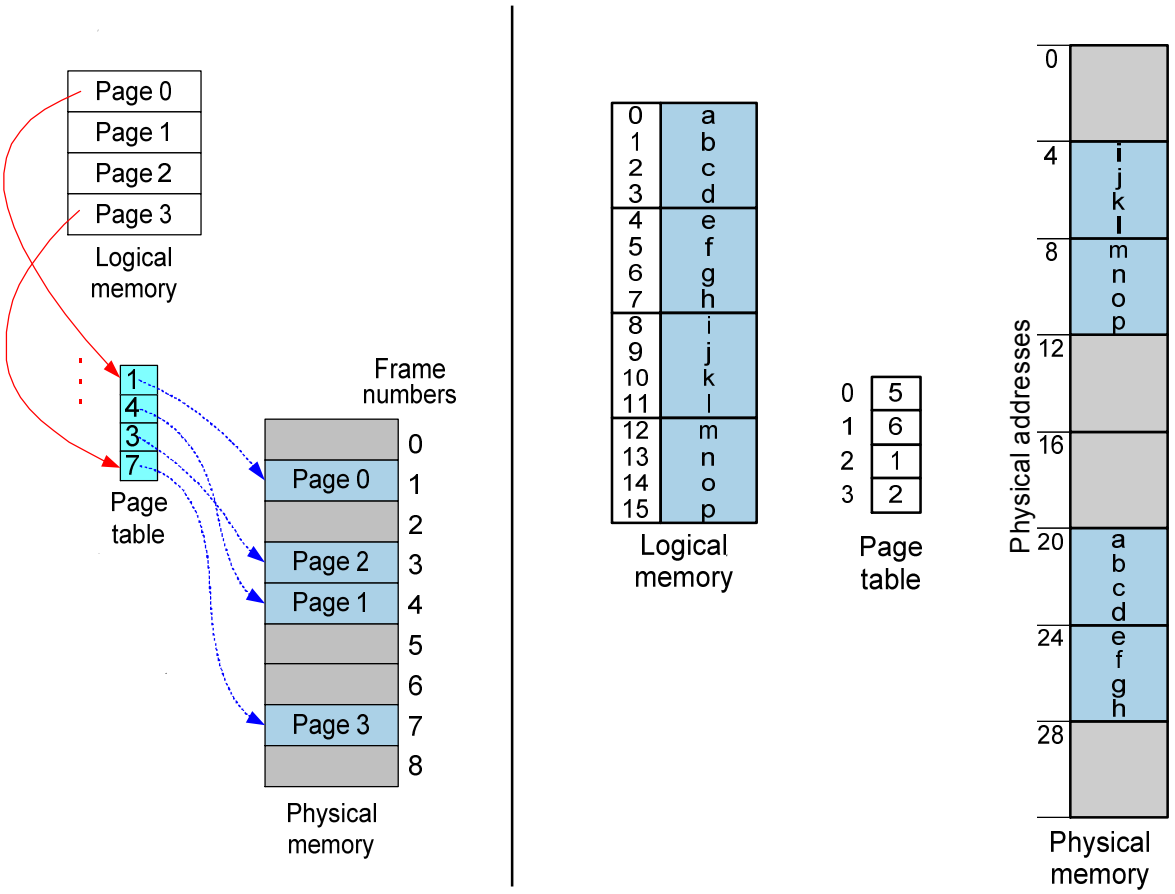
Fragmentation

- **External Fragmentation**
  - Total free memory space exists to satisfy a request, but it is not contiguous and contiguous space is required
- **Internal Fragmentation**
  - Memory is allocated using some fixed size memory "partitions"
  - Allocation size often defined by hardware
  - Allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
  - A compromise is needed:
    - Large partitions – too high internal fragmentation
    - Small partitions – too many of partitions to administer
- 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***
  - I/O problem
    - Latch job in memory while it is involved in I/O
    - Do I/O only into OS buffers

Paging

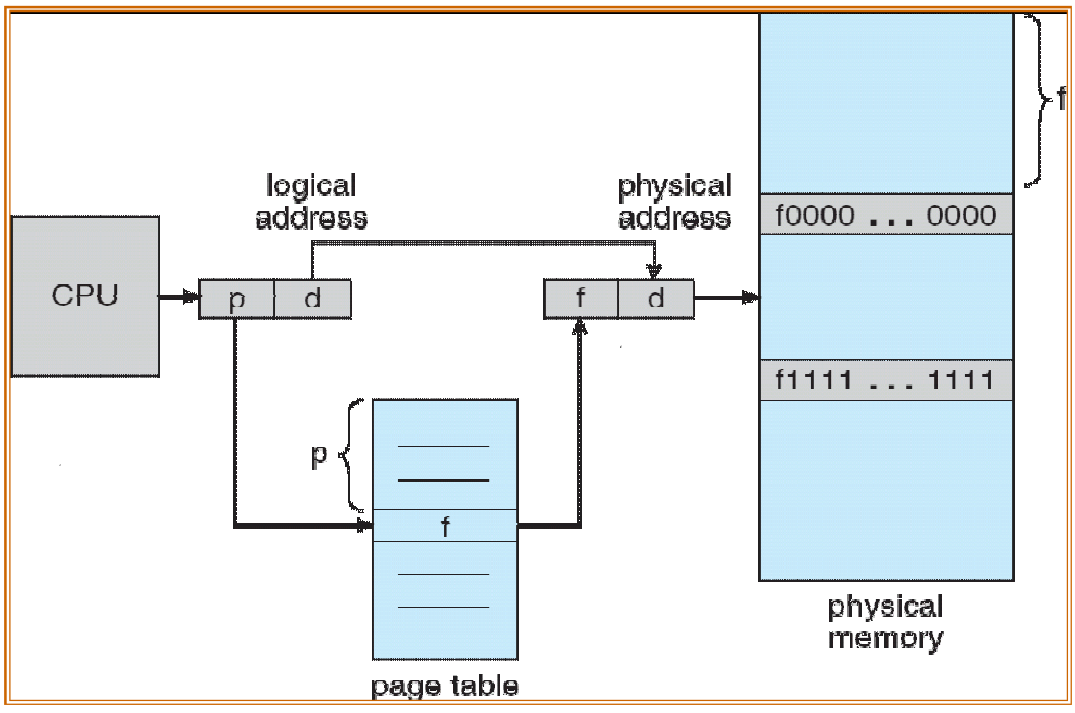
- ***Contiguous*** *logical address space* of a process can be mapped to ***noncontiguous*** *physical allocation*
  - process is allocated physical memory whenever the latter is available
- Divide physical memory into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 8192 bytes)
- Divide logical memory into blocks of same size called **pages**
- 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
- Internal fragmentation may occur

Paging Address Translation Principle



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

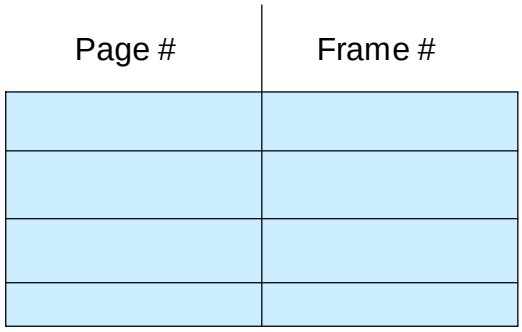


Implementation of Page Table

- Paging is implemented in hardware
- **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 **associative memory** or **translation look-aside buffers (TLBs)**

Associative Memory

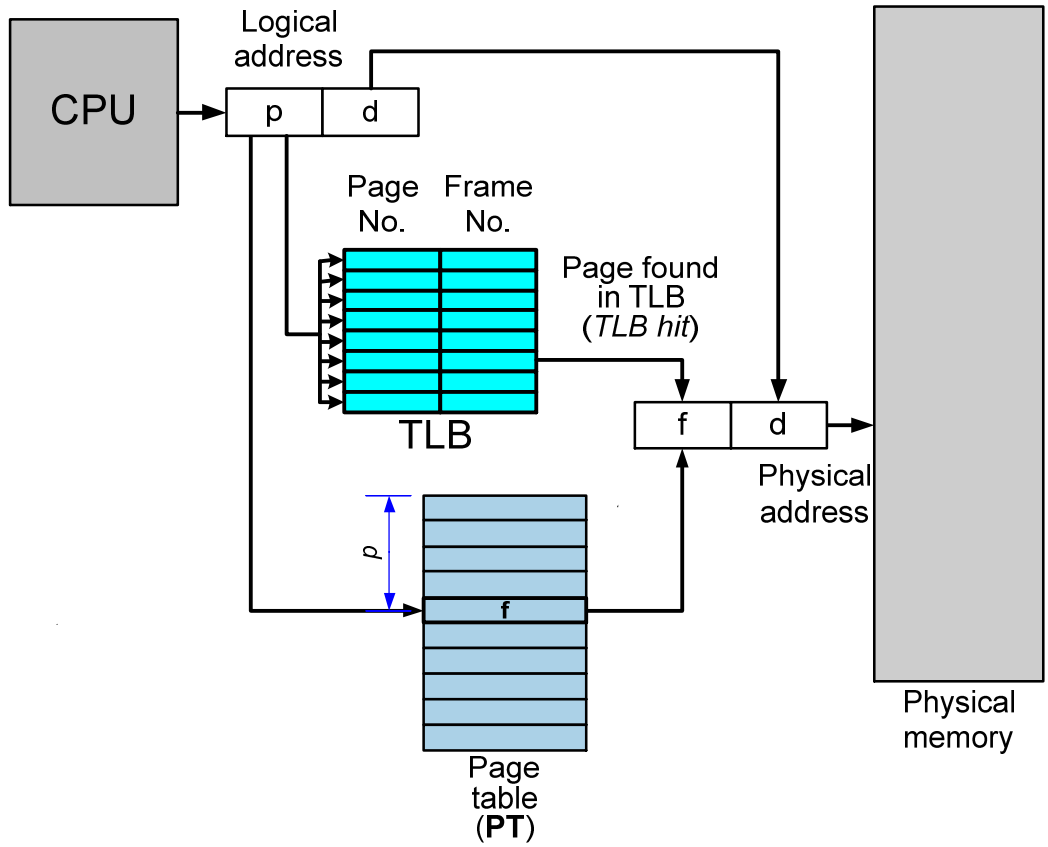
- Associative memory – parallel search – VERY COMPLEX CIRCUITRY



Address translation (P# → F#)

- If P# is in associative register, get F# out
- Otherwise get F# from page table in memory

Paging Hardware With TLB



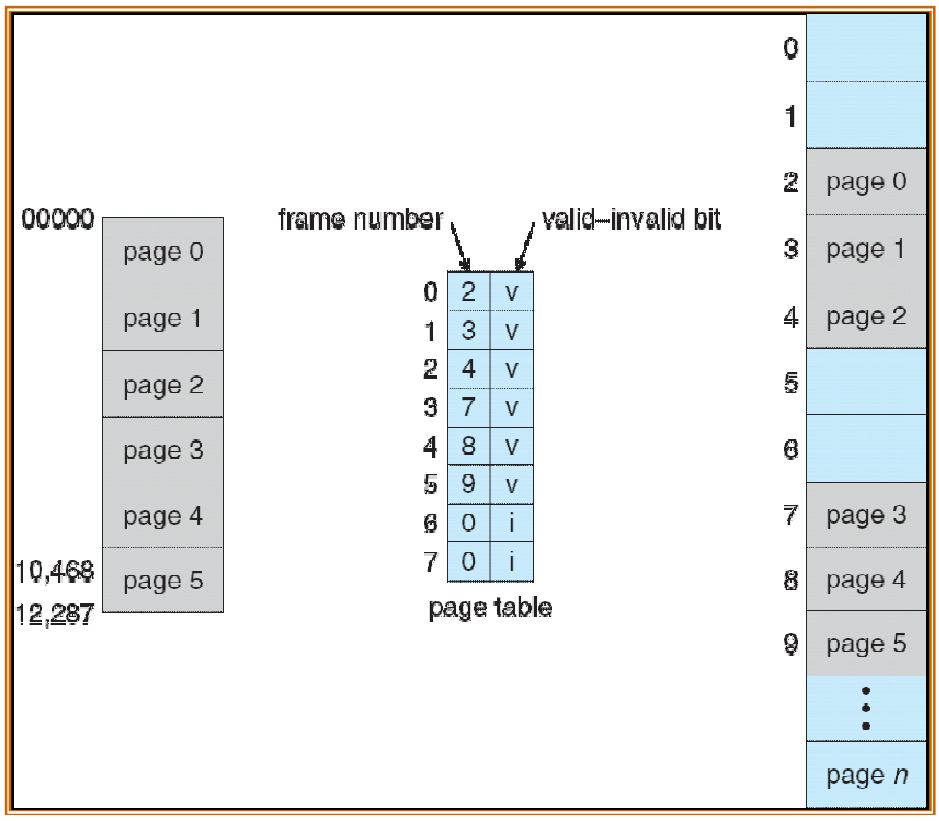
Paging Properties

- Effective Access Time with TLB
  - Associative Lookup = ε time unit
  - Assume memory cycle time is 1 microsecond
  - Hit ratio – percentage of times that a page number is found in the associative registers; ration related to number of associative registers, Hit ratio = α
  - **Effective Access Time** (EAT)

$$EAT = (1 + \varepsilon)\,\alpha + (2 + \varepsilon)(1 - \alpha)$$
$$= 2 + \varepsilon - \alpha$$

- **Memory protection** implemented by associating protection bit with each page
  - Usually two bits: Read-Only bit, Dirty bit (used as described later)
- **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

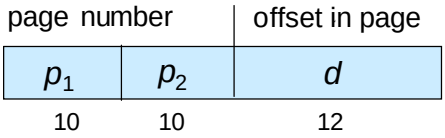


Page Table Structures

- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

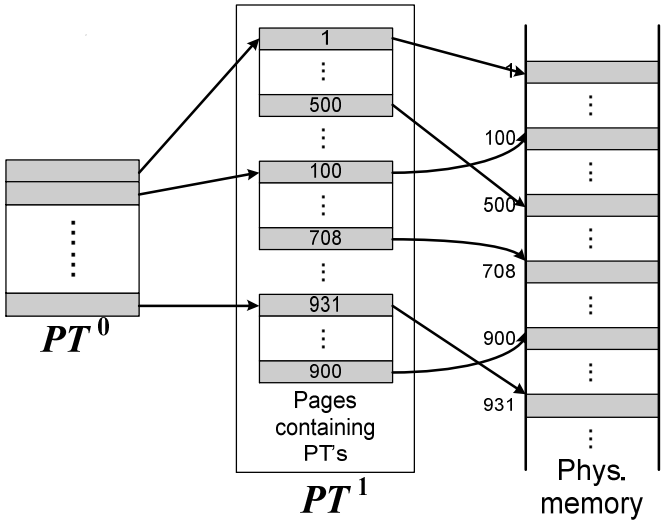
Hierarchical Page Tables

- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
  - A logical address (on 32-bit machine with 4K page size) is divided into:
    - a page number consisting of 20 bits
    - an offset within page consisting of 12 bits
  - The page table itself can also be paged, the page number can be further divided into:
    - a 10-bit page number
    - a 10-bit page offset
  - Thus, a logical address is:



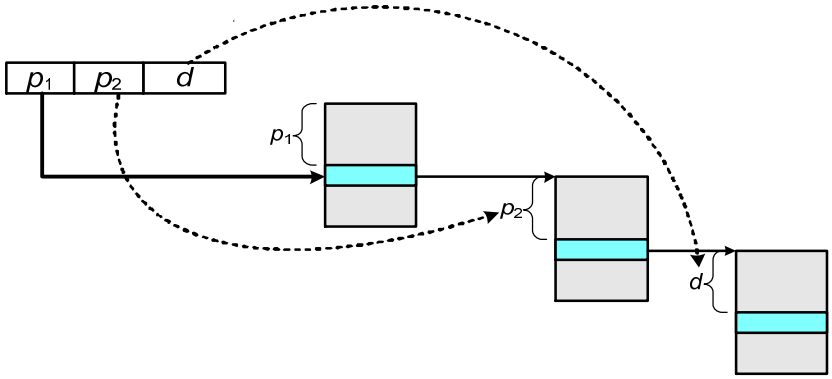
where p_i is an index into the *outer page table*, and p_2 is the displacement within the page of the outer page table

Two-Level Page-Table Scheme



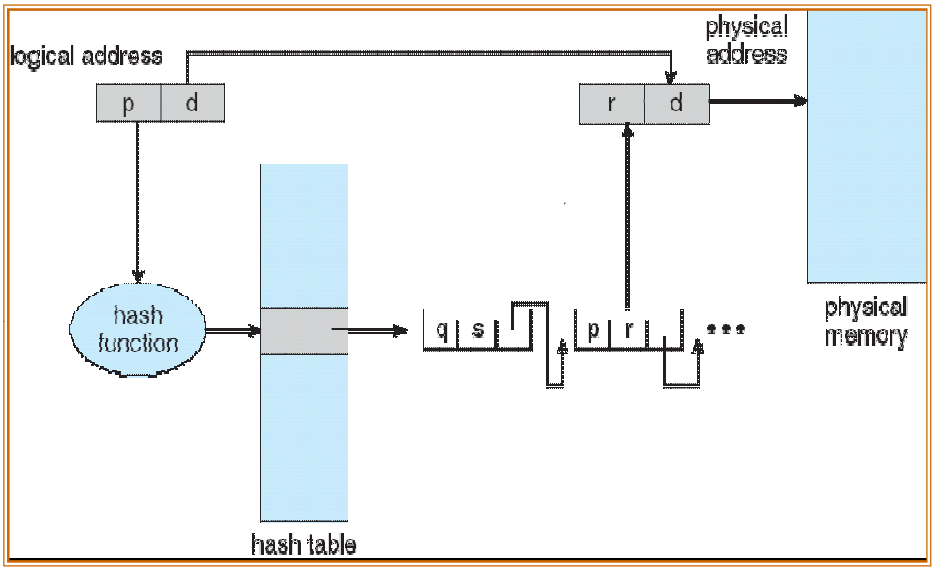
Two-level paging structure

Logical → physical address translation scheme



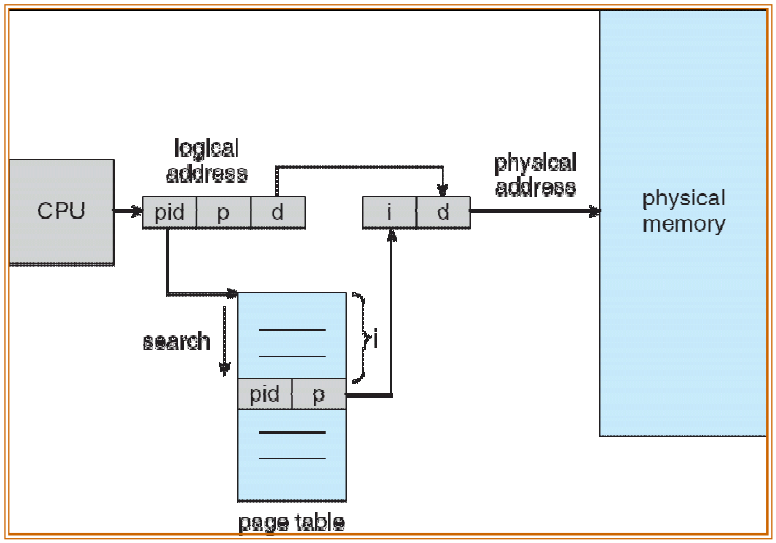
Hashed Page Tables

- Common for 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.



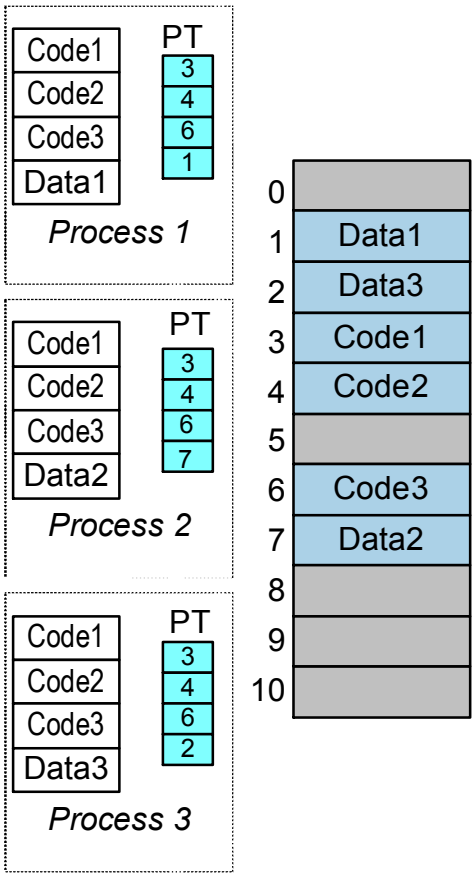
Inverted Page Table

- One entry for each frame of real 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
- Main advantage:
  - **only one PT for all processes**
- Use hash table to limit the search to one – or at most a few – page-table entries



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



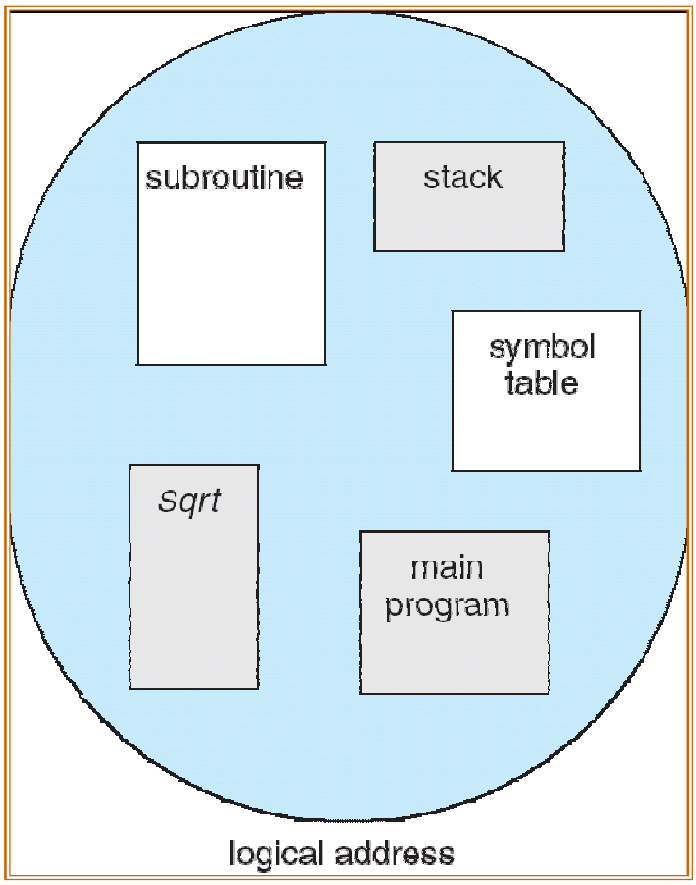
Three instances of a program

Segmentation

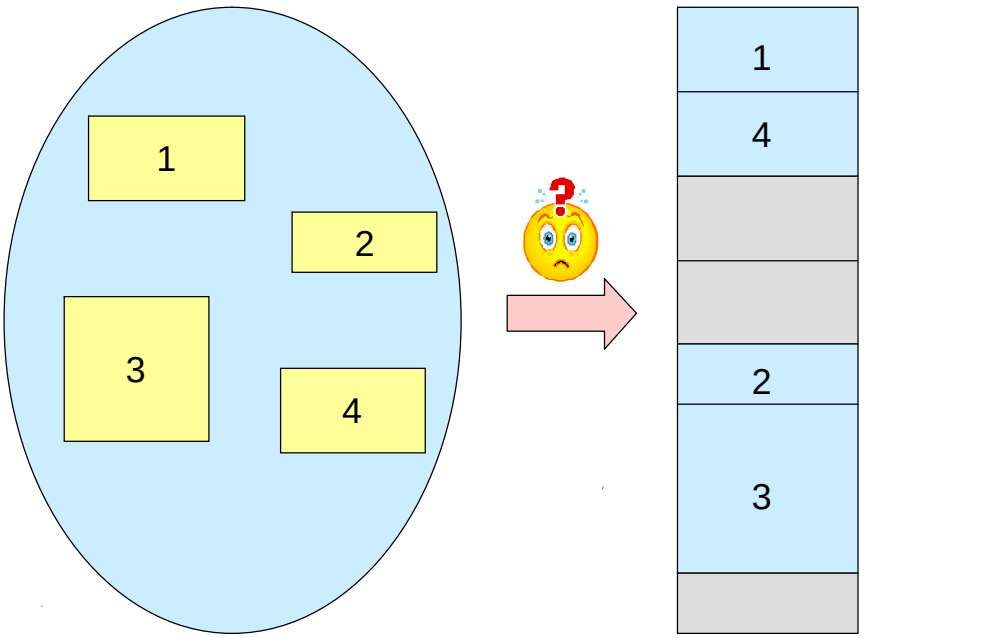
- Memory-management scheme that supports user view of memory
- A program is a collection of segments. A segment is a logical unit such as:

  main program,
  procedure,
  function,
  method,
  object,
  local variables, global variables,
  common block,
  stack,
  symbol table, arrays

User's View of a Program



Logical View of Segmentation



user space
= two dimensional view

physical memory space
= one dimensional structure

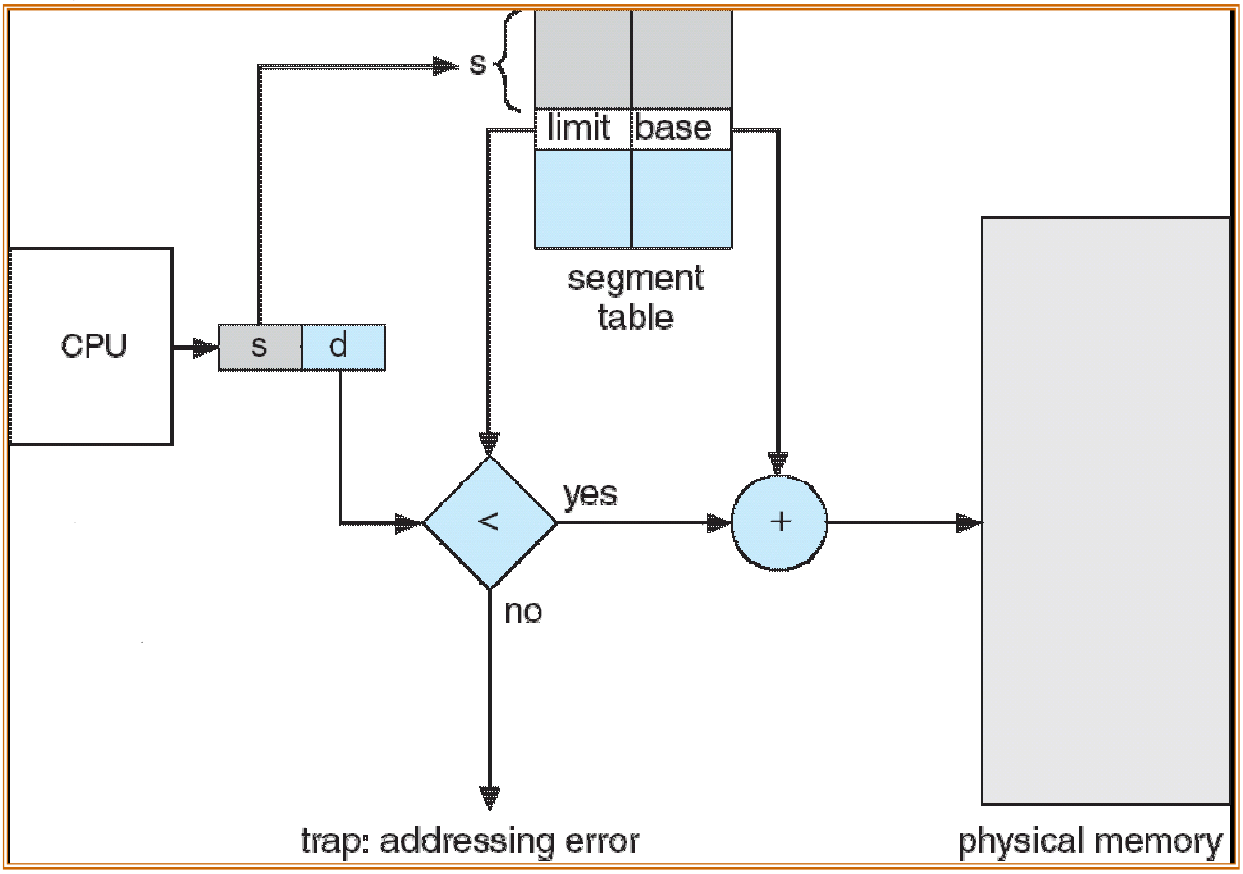
Segmentation Architecture

- Logical address consists of a couple:
  <segment-number, offset>,
- **Segment table** – maps two-dimensional physical addresses; each table entry has:
  - *base* – contains the starting physical address where the segments reside in memory
  - *limit* – specifies the length of the segment
- *Segment-table base register (STBR)* points to the segment table's location in memory
- *Segment-table length register (STLR)* indicates number of segments used by a program;
  segment number *s* is legal if *s* < STLR

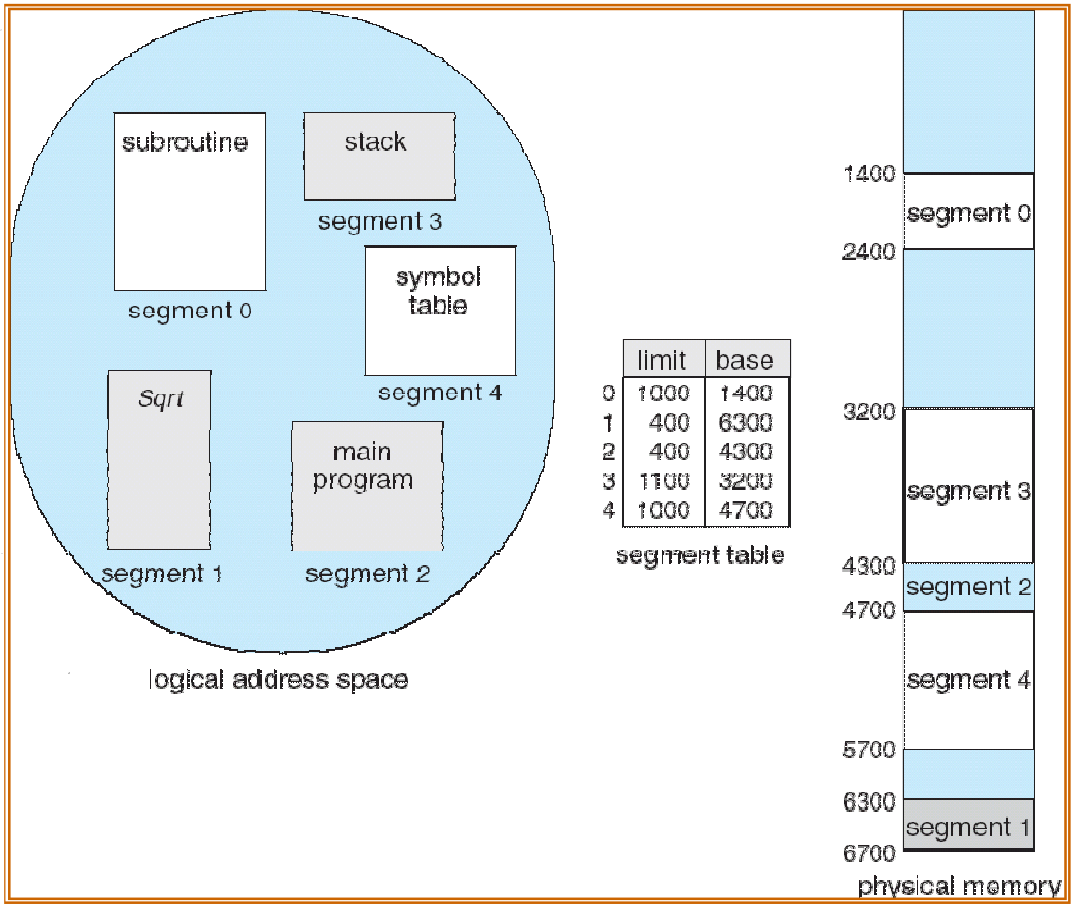
Segmentation Architecture (Cont.)

- **Relocation**
  - dynamic by segment table
- **Sharing**
  - shared segments – same segment number
- **Allocation**.
  - first fit/best fit – external fragmentation
- Protection. With each entry in segment table associate:
  - validation bit = 0 ⇒ illegal segment
  - read/write/execute privileges
- Protection bits associated with segments; code sharing occurs at segment level
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem
- A segmentation example is shown in the following diagram

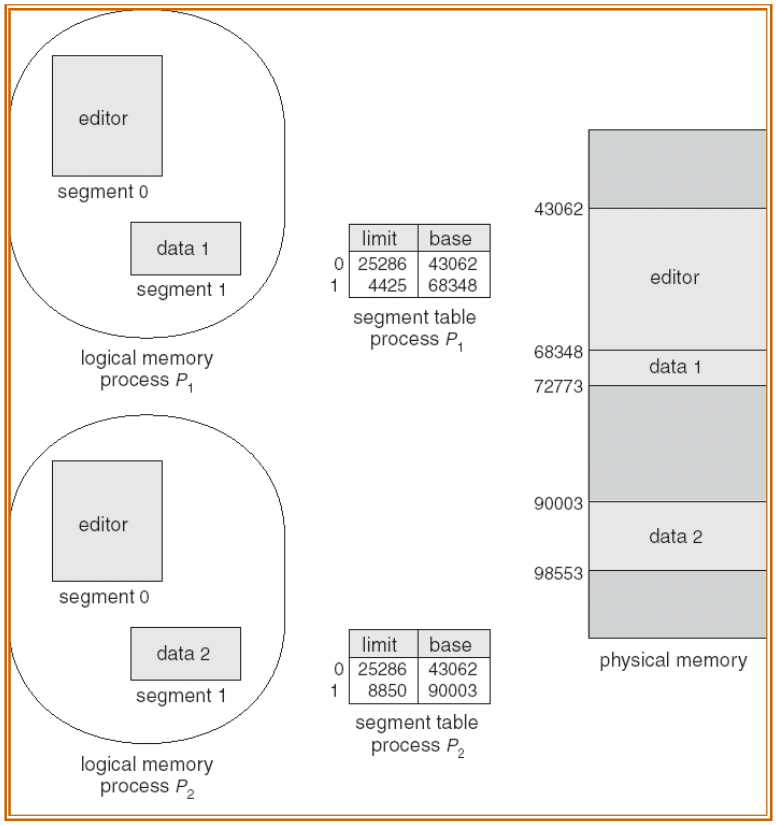
Address Translation Architecture



Example of Segmentation

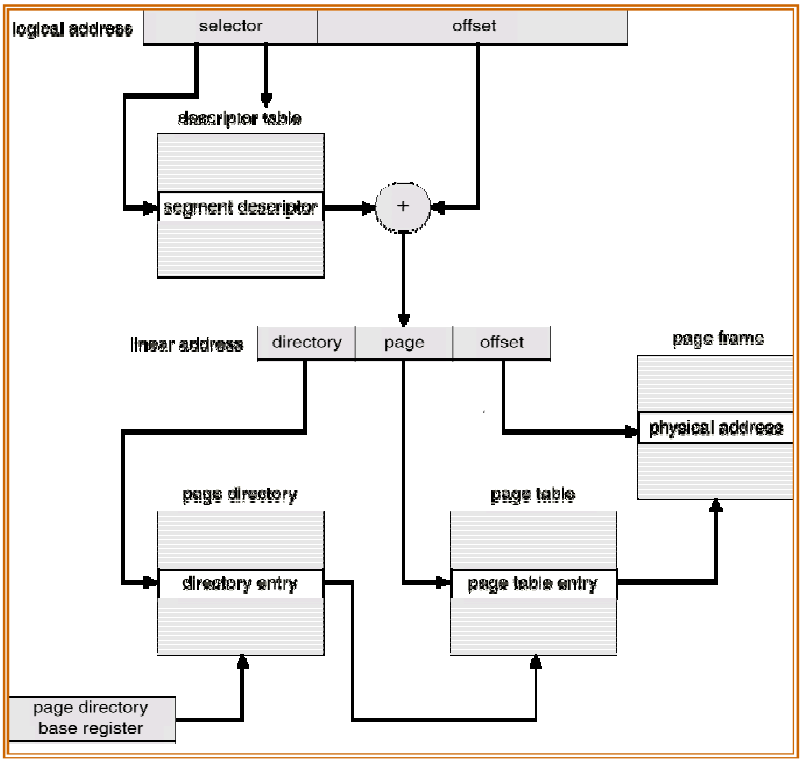


Sharing of Segments



Segmentation with Paging – Intel IA32

- IA32 architecture
  - uses segmentation with paging for memory management with a two-level paging scheme



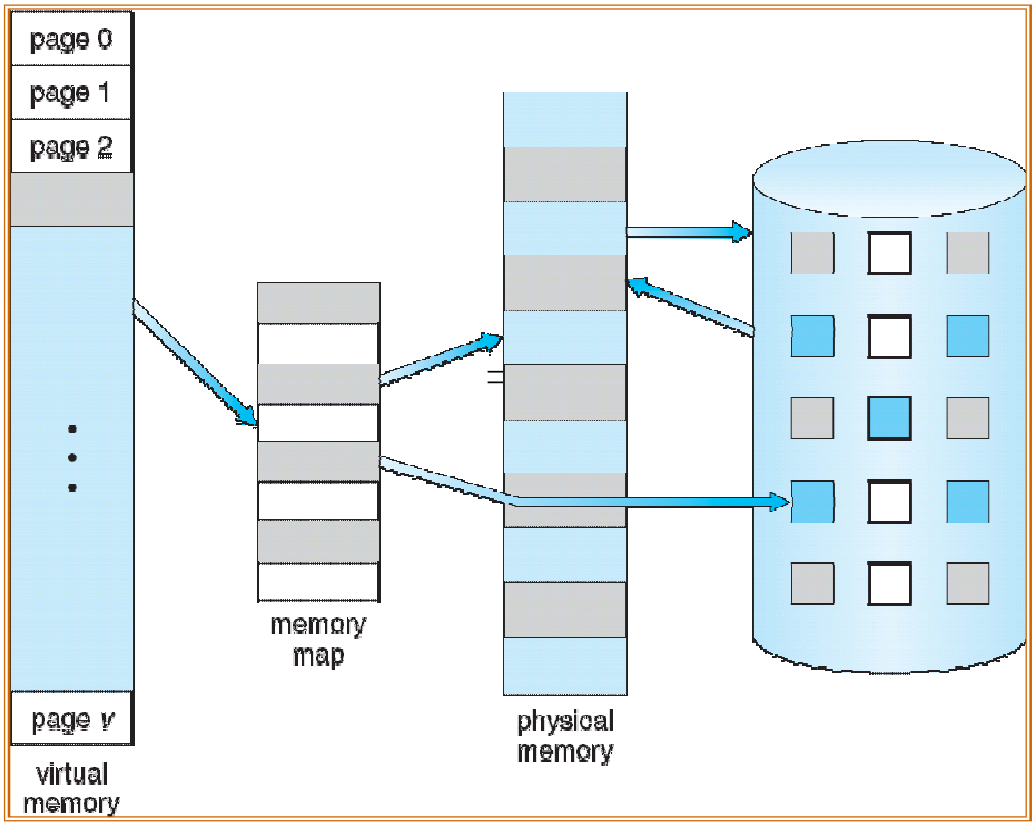
Linux on Intel IA32

- Uses minimal segmentation to keep memory management implementation more portable
- Uses 6 segments:
  - Kernel code
  - Kernel data
  - User code (shared by all user processes, using logical addresses)
  - User data (likewise shared)
  - Task-state (per-process hardware context)
  - LDT
- Uses 2 protection levels:
  - Kernel mode
  - User mode

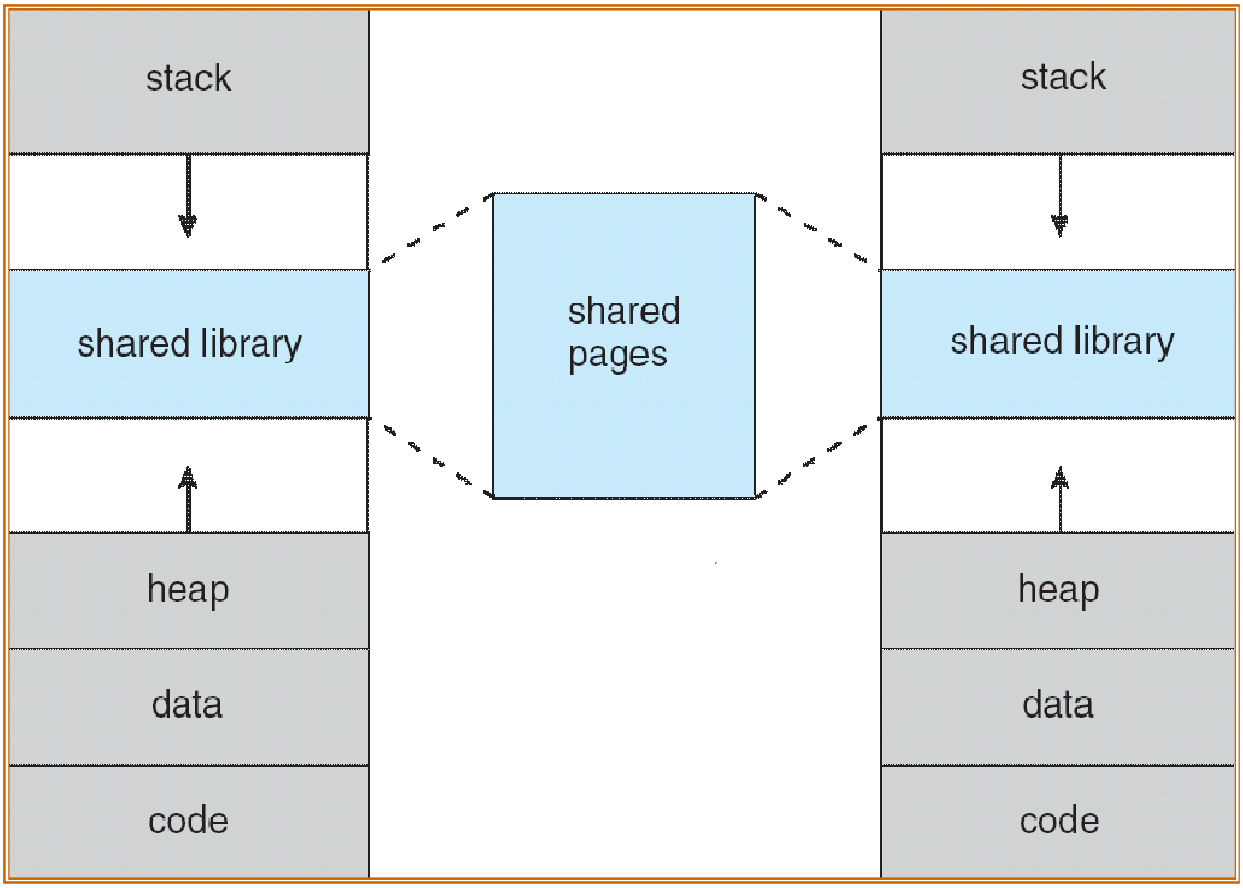
Virtual Memory Background

- **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.
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

If Virtual Memory is Larger Than Physical Memory

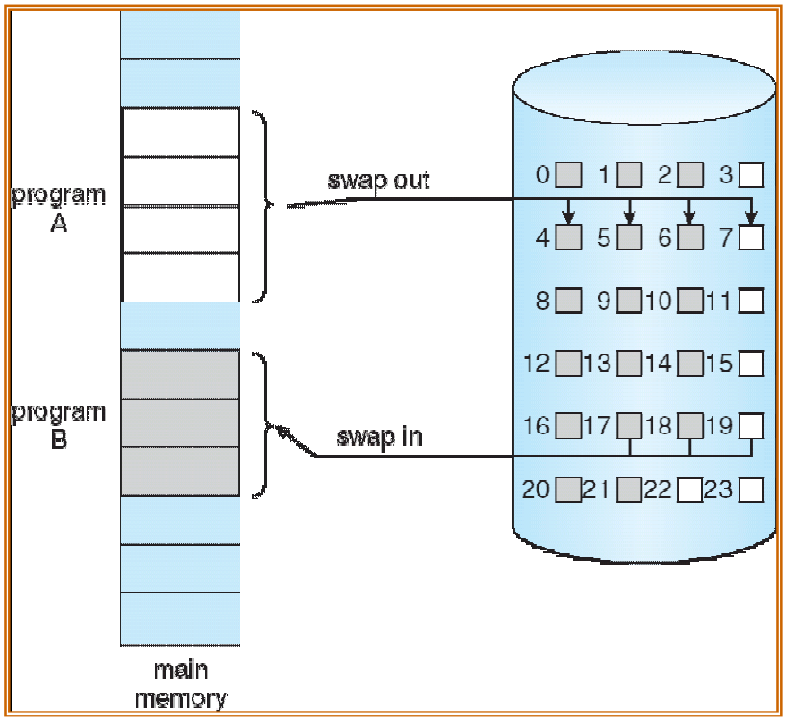


Shared Library Using Virtual Memory



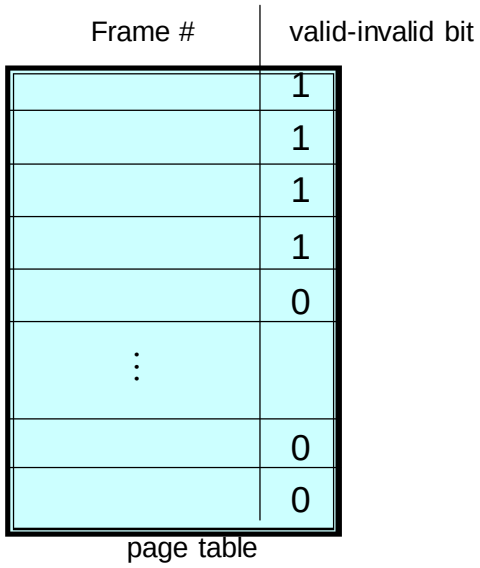
Demand Paging

- Bring a page into memory only when it is needed
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users
- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory



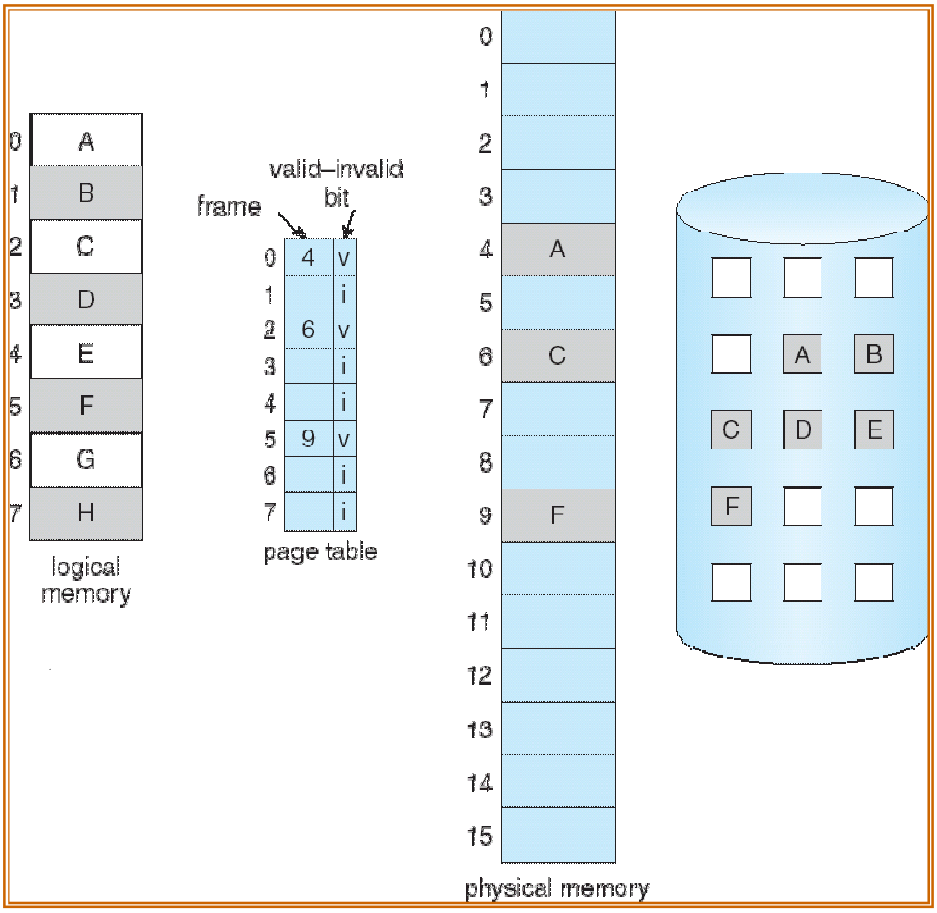
Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated
  (1 ⇒ in-memory, 0 ⇒ not-in-memory)
- Initially valid–invalid but is set to 0 on all entries
- Example of a page table snapshot:



- During address translation, if valid–invalid bit in page table entry is 0 ⇒ page fault

Page Table when Some Pages are Not in Main Memory

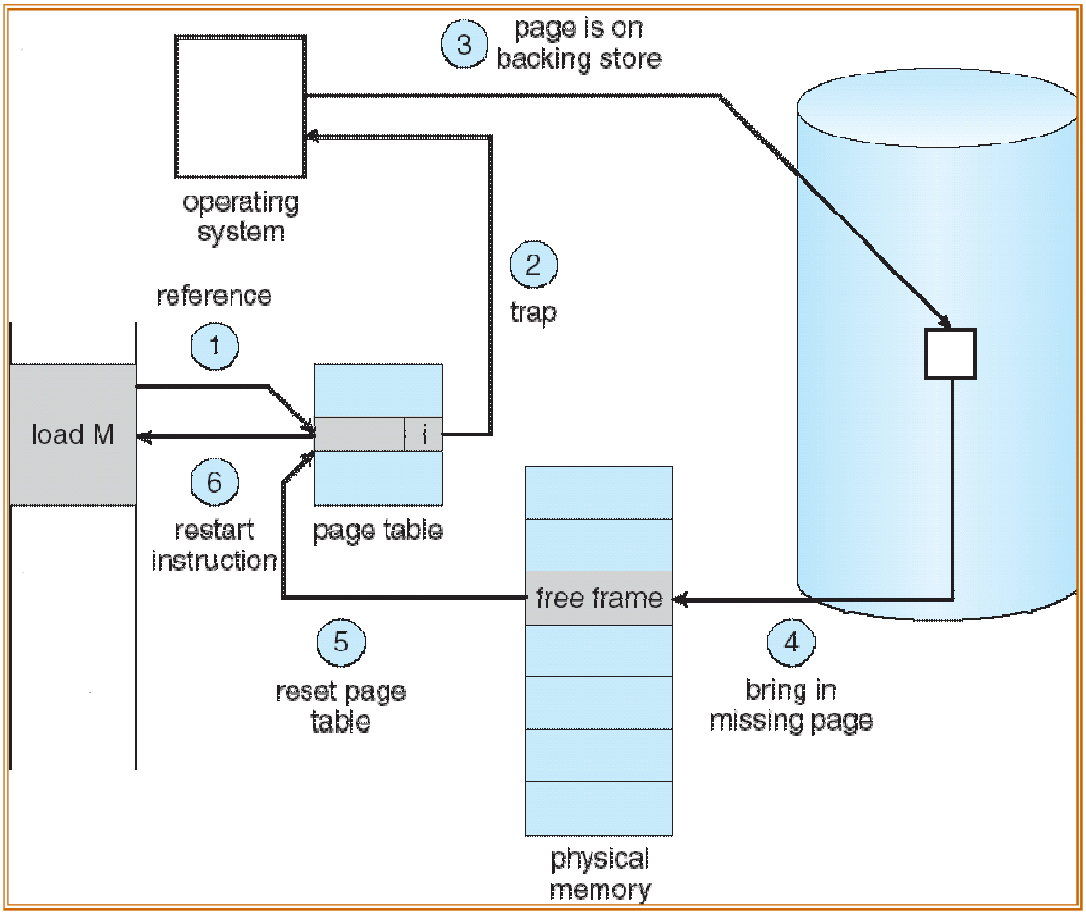


Page Fault

1. If there is a reference to a page with invalid bit, it will **trap** to OS ⇒ **page fault**
2. OS looks at another table (usually in PCB) to decide if:
   - Really invalid reference ⇒ abort the offending process
   - Just page not in memory
3. Find a free frame
4. Swap-in page into frame
5. Modify page table, validation bit = 1
6. Restart instruction

- Pure demand paging:
  - Prepare the process' "memory image" in the disk swap area
  - Start process with all pages invalid

Steps in Handling a Page Fault



What Happens if There is no Free Frame?

- **Page replacement** – find some page in memory, but not really in use, swap it out (if needed)
  - Which page? (algorithm)
  - Performance – want an algorithm which will result in minimum number of page faults
- Consequence: Same page may have to be brought into memory many times

Performance of Demand Paging

- Page Fault Rate $0 \le p \le 1.0$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault
- Effective Access Time (EAT)

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

- Example:
  - Memory access time = 1 μsec
  - 50% of the time the page that is being replaced has been modified and therefore needs to be swapped out
  - Swap Page Time = 10 msec = 10,000 μsec

$$\begin{aligned} EAT &= (1 - p) * 1 + p * (15{,}000) \\ &= 1 + 14{,}999\,p \qquad [\text{in } \mu\text{sec}] \end{aligned}$$

Process Creation

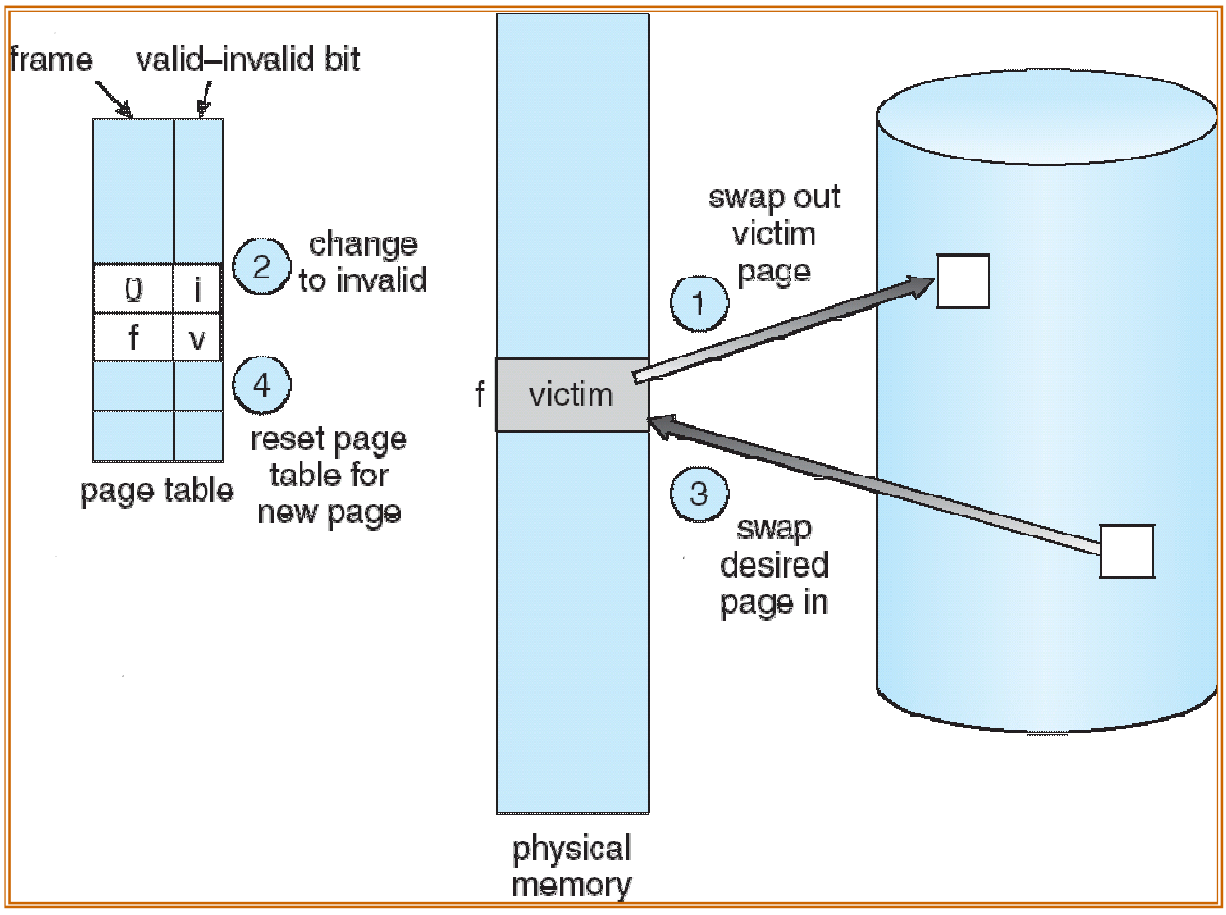
- Virtual memory allows other benefits during process creation:
  - Copy-on-Write
  - Memory-Mapped Files (later)
- POSIX `fork()` – very inefficient
  - Need to duplicate (copy) all process memory space
  - Not necessary with virtual memory
- **Copy-on-Write** (**COW**)
  - Allows both parent and child 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
  - Free pages are allocated from a pool of zeroed-out pages

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 swapped-out (written) to disk
- Page replacement completes separation between logical memory and physical memory
  - large virtual memory can be provided on a smaller physical memory

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

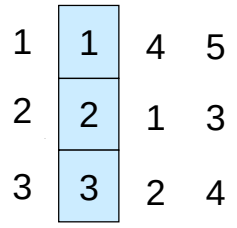


Page Replacement Algorithms

- Want lowest page-fault rate
- Evaluate algorithm by running it on a particular string of memory references (*reference string*) and computing the number of page faults on that string
- In all our examples, the reference string is

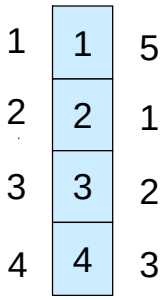
  1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- First-In-First-Out (FIFO) Algorithm

3 frames



9 page faults

4 frames



10 page faults

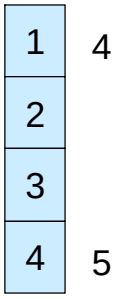
- FIFO Replacement
  - more frames ⇒ more page faults - Belady's Anomaly

The Optimal Algorithm

- Replace page that will not be used for longest period of time in future
  - fiction
- 4 frames example

  1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

4 frames

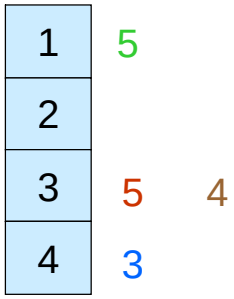


6 page faults

- How do we know this? We cannot
- Nothing better exists
  - Used to measure how well our algorithm performs

Least Recently Used (LRU) Algorithm

- Approximation to optimal algorithm
- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

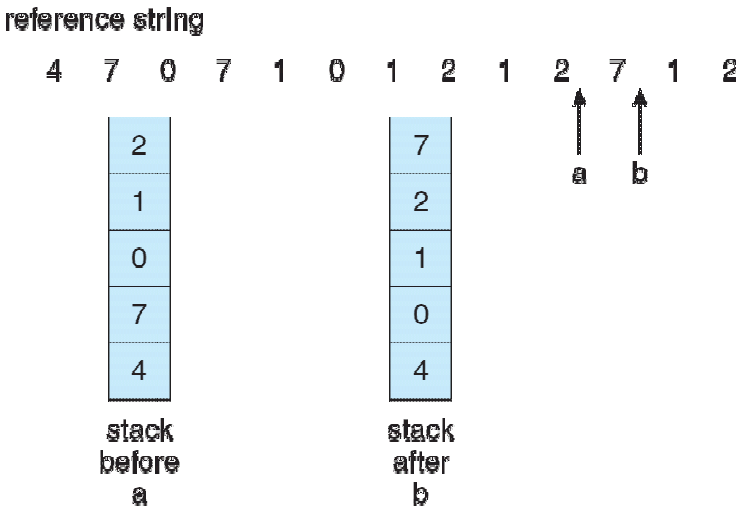


7 page faults

- Time-stamp implementation
  - Every page entry has a time-stamp field; every time the page is referenced, copy the system clock into this field
    - Time demanding
    - Page-table grows
  - When a page needs to be changed, look at the time-stamps to determine which is the victim candidate and select the one with the oldest time-stamp
    - Expensive search

LRU Algorithm (Cont.)

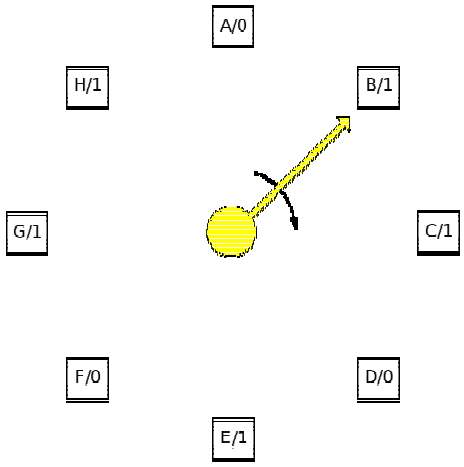
- Stack implementation – keep a stack of page numbers in a double linked list:
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed
  - No search for replacement
  - Difficult house-keeping on every page reference
    - Should be done by hardware for efficiency



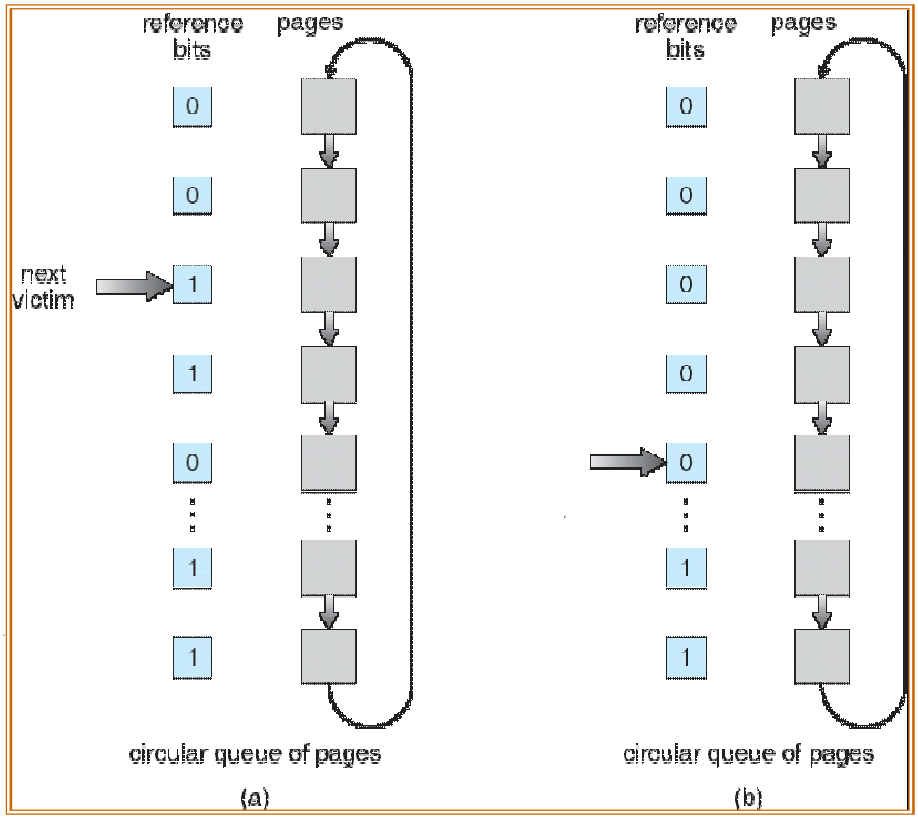
- Conclusion:
  - Both Time-stamp and Stack implementations are too complicated

LRU Approximation Algorithms

- Reference (access) bit (*a*-bit)
  - With each page associate an *a*-bit, on swap-in set *a*=0
  - When page is referenced set *a*=1
  - Replace the page with *a* == 0 (if one exists)
  - Cheap, but we do not know the proper page search order
- Second chance (Clock replacement)
  - Uses the *a*-bit, list of swapped-in pages and a clock face
  - Every page reference adds a "life", every clock-hand pointing removes a "life"
  - Victim is the page pointed by the hand that has no life
  - Example
  - On page-fault the page pointed by the clock hand is examined
    - if *a*==0 make this page the victim
    - if *a*==1 then *a*=0, move the hand and examine the next pointed page
  - The numeric simulations show very good approximation of the real LRU



Second-Chance (Clock) Page-Replacement Algorithm



Allocation of Frames

- Each process needs some *minimum* number of pages to execute
- Example: IBM 370 – 6 pages to handle *MOVE from, to* instruction:
  - instruction is 6 bytes $\Rightarrow$ might span 2 pages
  - 2 pages to handle *from*
  - 2 pages to handle *to*
  - **3 page faults can occur for this single instruction!**
- Two major allocation schemes
  - Fixed allocation
  - Priority allocation

Allocation Strategies

- Equal allocation
  - For example, if there are 100 frames and 5 processes, give each process 20 frames.
- Proportional allocation
  - Allocate according to the size of process

$$s_i = \text{size of process } p_i$$

$$S = \sum s_i$$

$$m = \text{total number of frames}$$

$$a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m$$

2 processes example

$m = 64$

$s_1 = 10$

$s_2 = 127$

$a_1 = \frac{10}{137} \times 64 \approx 5$

$a_2 = \frac{127}{137} \times 64 \approx 59$

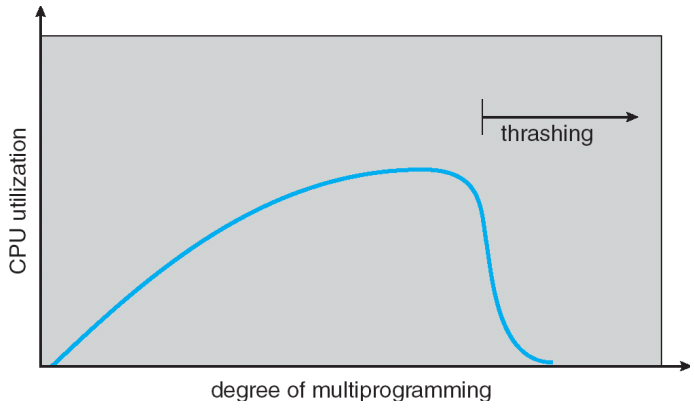
- Priority Allocation
  - Use a proportional allocation scheme using priorities rather than size
  - If process p_i generates a page fault,
    - select for replacement one of its frames, or
    - select for replacement a frame from a process with lower priority

Global vs. Local Allocation

- **Global replacement**
  - process selects a replacement frame from the set of all frames
  - one process can take a frame from another
- **Local replacement**
  - each process selects from only its own set of allocated frames

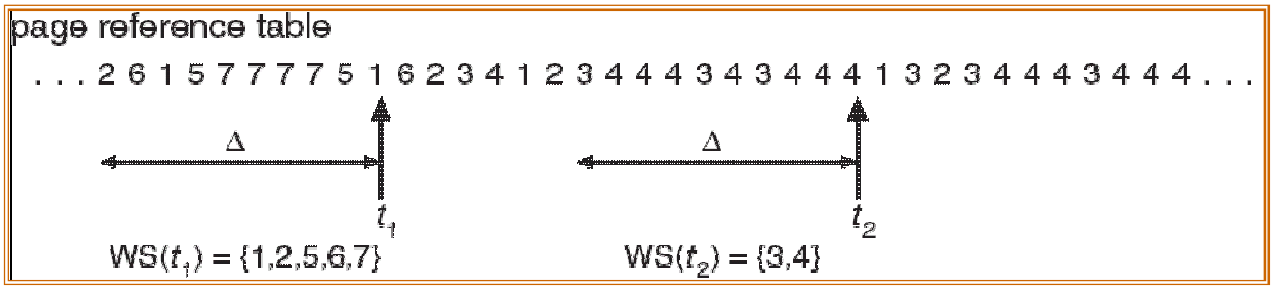
Memory Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high. This leads to:
  - low CPU utilization
  - operating system thinks that it may increase the degree of multiprogramming (number of processes)
- **Thrashing** ≡ a process is busy swapping pages in and out and does not compute
- Why does demand paging work properly?
- Locality model
  - Process migrates from one locality to another
  - Localities may overlap
- Why does thrashing occur?
  Σ size of locality > total memory size = *m*



Working-Set Model

- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  Example: 10,000 instructions
- WSS_i (working set of process p_i) = total number of pages referenced in the most recent Δ (varies in time)
  - if Δ small will not encompass entire locality
  - if Δ large will encompass several localities
  - if $\Delta \rightarrow \infty \Rightarrow$ will encompass entire program
- $D = \Sigma\ WSS_i \equiv$ total demanded frames
- if $D > m \Rightarrow$ Thrashing
- Policy: if $D > m$, then suspend one or more processes

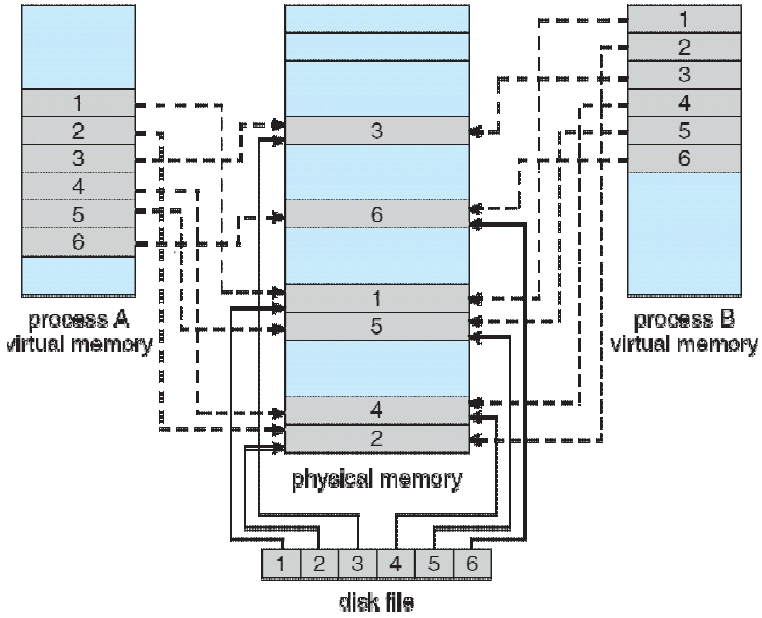


Keeping Track of the Working Set

- Approximate with interval timer + a reference bit
- Example: Δ = 10,000
  - Timer interrupts after every 5000 memory references
  - Keep in memory 2 “indicator” bits for each page
  - Whenever a timer interrupts copy reference bit to “indicators” and reset the values of all reference bits
  - If one of the bits in memory == 1 assume page in working set
- Better accuracy:
  - 10 bits and interrupt every 1000 memory refs
  - Very expensive

Memory-Mapped Files

- Memory-mapped file I/O allows file I/O to be treated as routine memory access by **mapping** a disk block to a page in memory
- A file is initially read using demand paging. A page-sized portion of the file is read from the file system into a physical page. Subsequent reads/writes to/from the file are treated as ordinary memory accesses.
- Simplifies file access by treating file I/O through memory rather than **read() write()** system calls
- Also allows several processes to map the same file allowing the pages in memory to be shared



Prepaging

- Prepaging
  - To reduce the large number of page faults that occurs at process startup
  - Prepage all or some of the pages a process will need, before they are referenced
  - But if prepaged pages are unused, I/O and memory was wasted
  - Assume *s* pages are prepaged and *α* of the pages is used
    - Is cost of $s * \alpha$ save pages faults > or < than the cost of prepaging $s * (1 - \alpha)$ unnecessary pages?
    - α near zero $\Rightarrow$ prepaging loses

Virtual Memory & Program Structure

- Program structure

  ```
  double[512][512] data;
  ```

  - Each row is stored in one 4 KB page
  - Program 1

    ```
    for (j = 0; j <512; j++)
      for (i = 0; i < 512; i++)
        data[i][j] = 0;
    ```

    Potentially 512 x 512 = 262,144 page faults

  - Program 2

    ```
    for (i = 0; i < 512; i++)
      for (j = 0; j < 512; j++)
        data[i][j] = 0;
    ```

    512 page faults

- Good to know how data is stored

Windows XP

- Uses demand paging with **clustering**. Clustering swaps in pages surrounding the faulting page.
- Processes are assigned **working set minimum** and **working set maximum**
- Working set minimum is the minimum number of pages the process is guaranteed to have in memory
- A process may be assigned as many pages up to its working set maximum
- When the amount of free memory in the system falls below a threshold, **automatic working set trimming** is performed to restore the amount of free memory
- Working set trimming removes pages from processes that have pages in excess of their working set minimum

End of Lesson 6

Questions?