Memory Management
Virtual Memory

Background; key issues

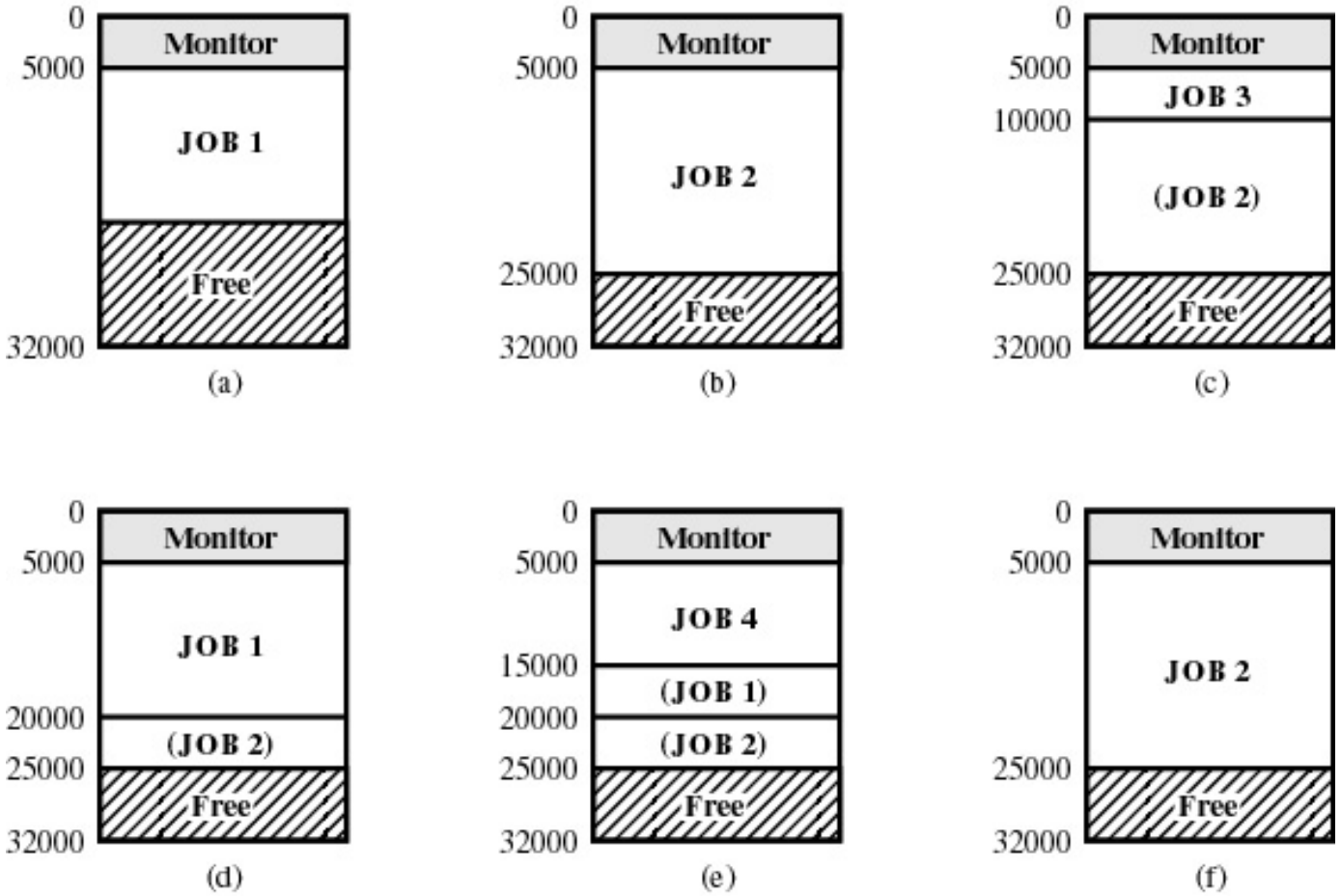
Memory allocation schemes

Virtual memory

Memory management design and implementation issues

Remember…
Basic OS structures: intro in historical order
(step 2 & 2,5) multiprogramming needs …

- … memory management!



Background

- Program
  - must be brought into memory (must be made a process) to be executed.
  - process might need to wait on the disk, in *input queue* before execution starts
- Memory
  - can be subdivided to accommodate multiple processes
  - needs to be allocated efficiently to pack as many processes into memory as possible

Memory Management

- Ideally programmers want memory that is
  - large
  - fast
  - non volatile
- Memory hierarchy
  - small amount of fast, expensive memory – cache
  - some medium-speed, medium price main memory
  - gigabytes of slow, cheap disk storage
- Memory manager handles the memory hierarchy

The position and function of the MMU

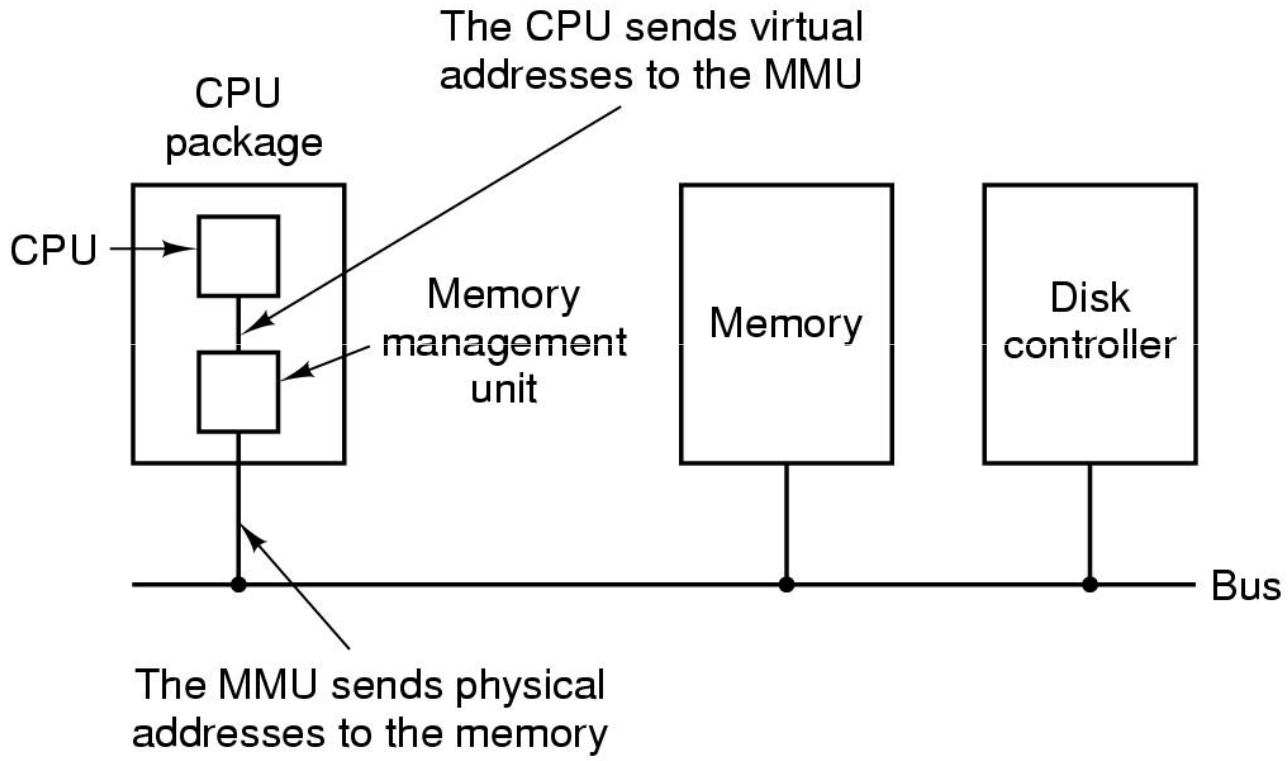
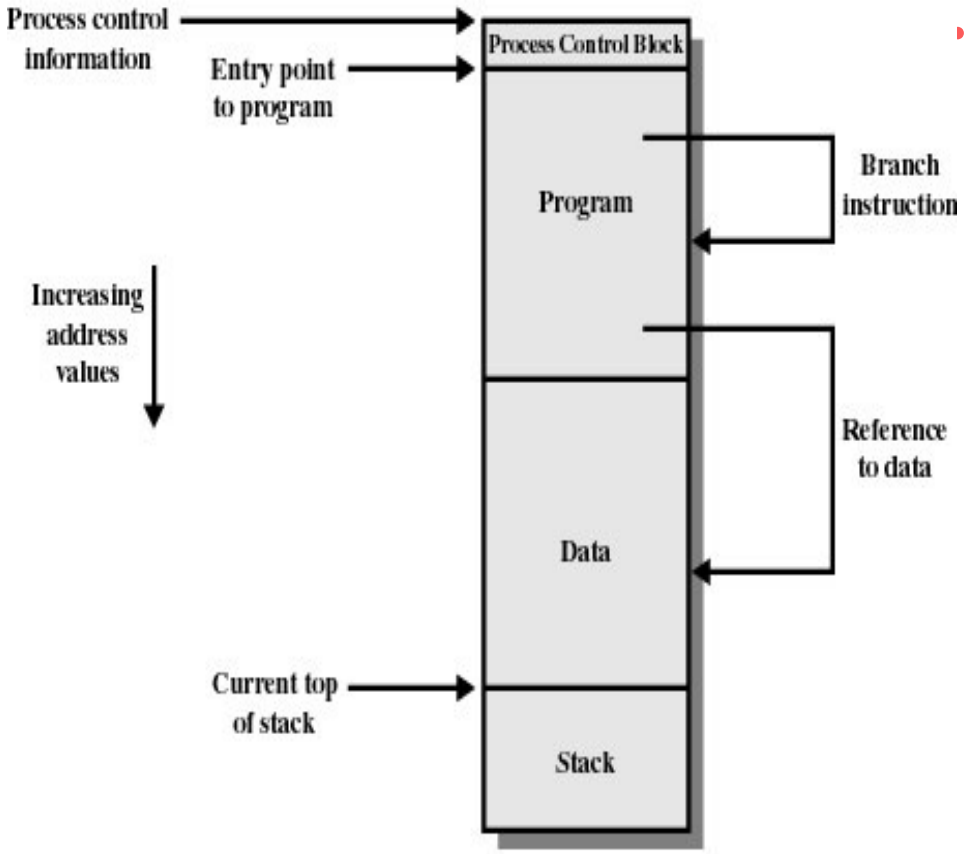


Fig. source: (A.T. MOS 2/e)

Relocation and Protection

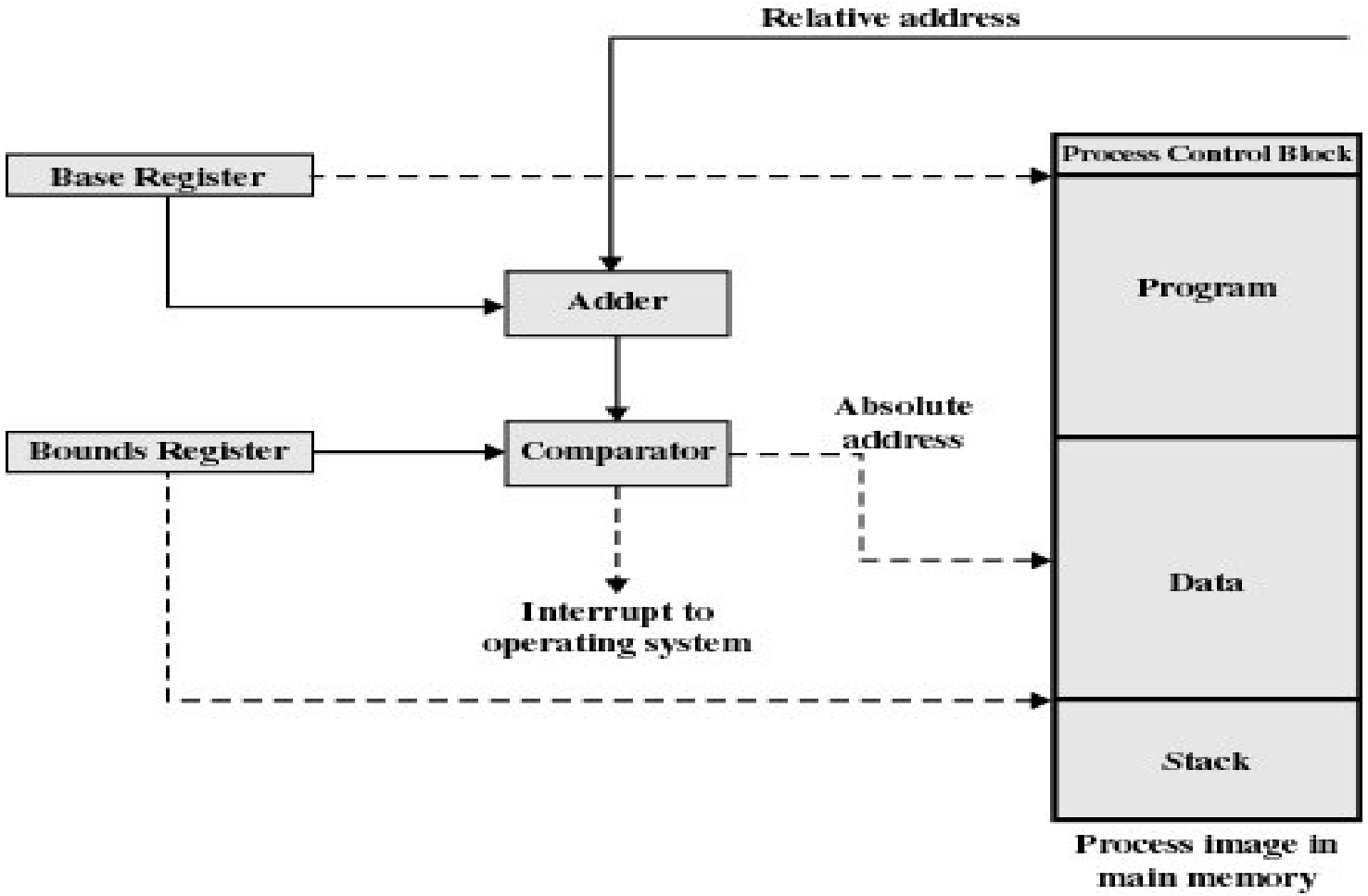


- Cannot be sure where program will be loaded in memory
  - address locations of variables, code routines cannot be absolute
  - must keep a program out of other processes' partitions

Figure 7.1 Addressing Requirements for a Process

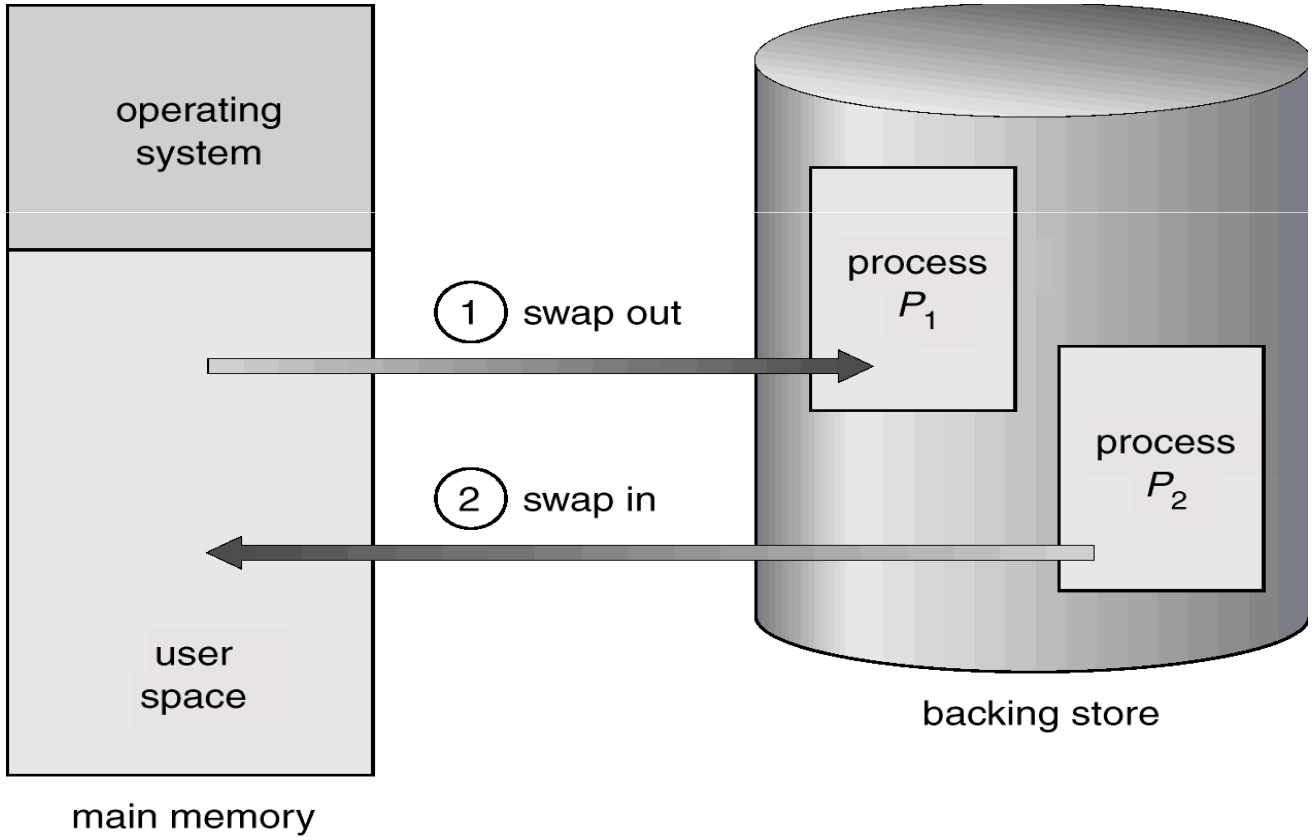
Hardware support for relocation and protection

Base, bounds registers: set when the process is executing



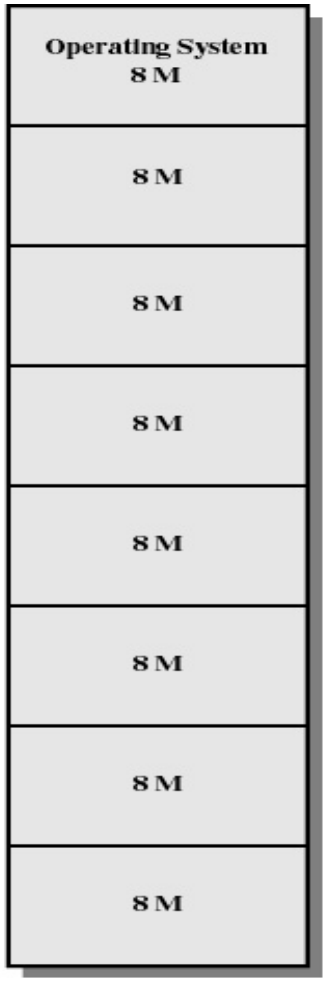
Swapping (Suspending) a process

A process can be *swapped out* of memory to a *backing store (swap device)* and later brought back (*swap-in*) into memory for continued execution.

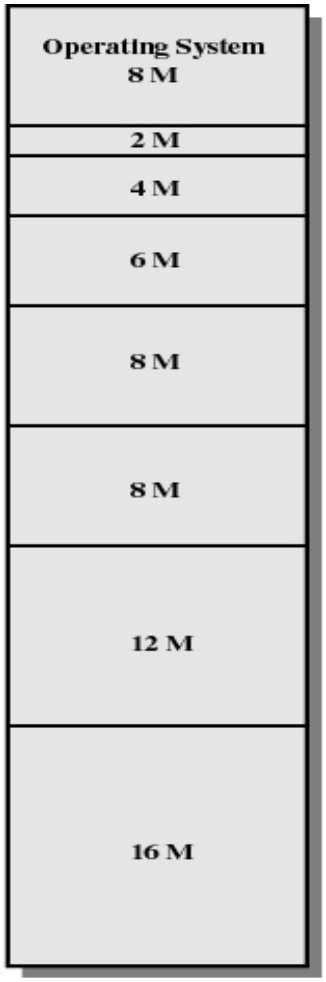


How is memory allocated?

Contiguous Allocation of Memory: Fixed Partitioning



(a) Equal-size partitions



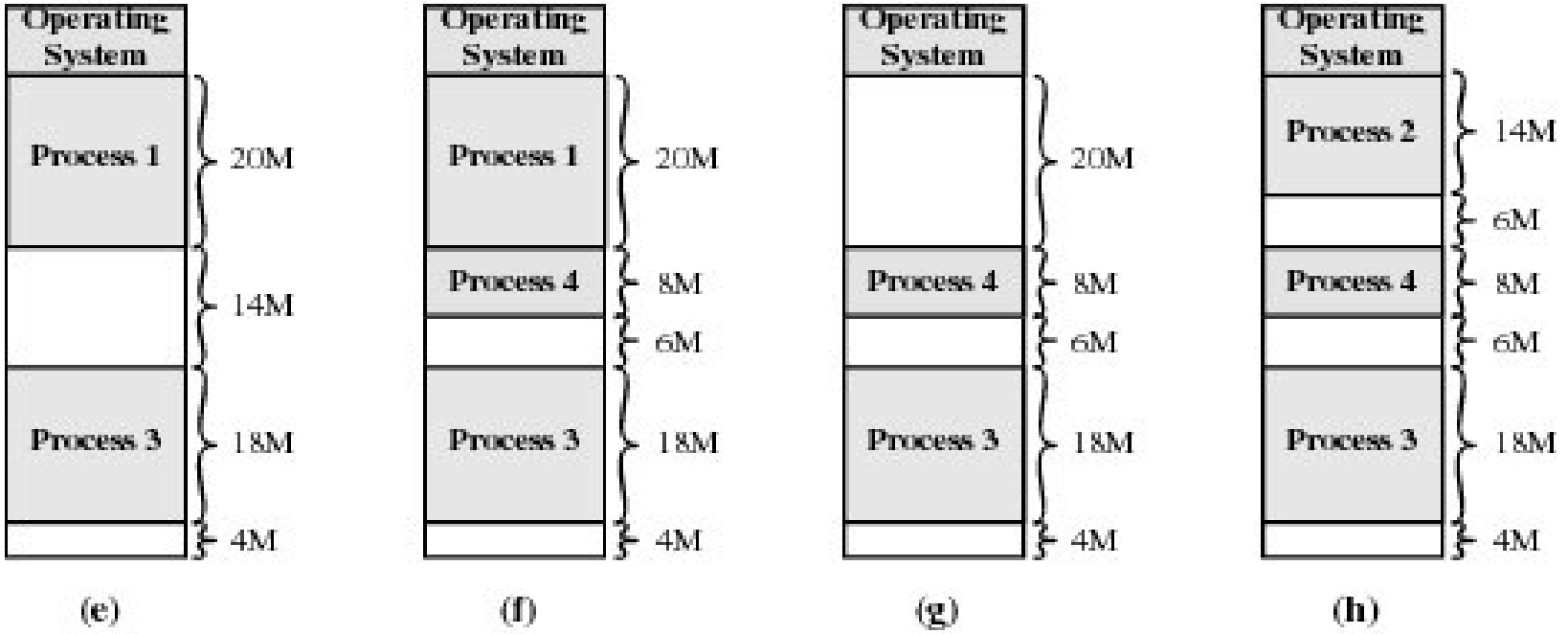
(b) Unequal-size partitions

- any program, no matter how small, occupies an entire partition.
- this causes **internal fragmentation**.

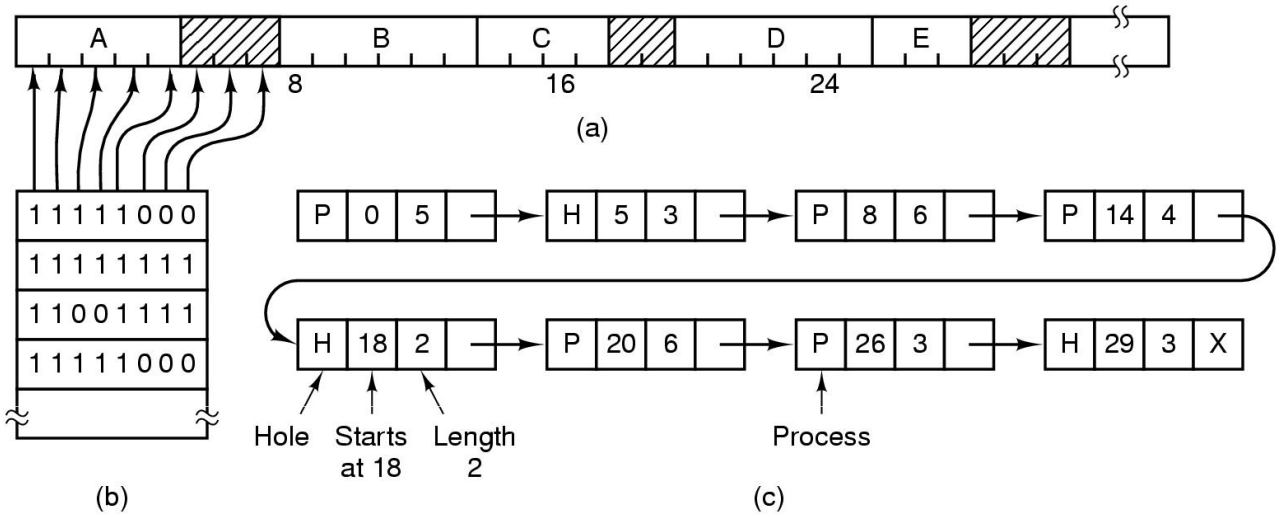
Figure 7.2 Example of Fixed Partitioning of a 64-Mbyte Memory

Contiguous Allocation: Dynamic Partitioning

- Process is allocated exactly as much memory as required
- Eventually holes in memory: **external fragmentation**
- Must use **compaction** to shift processes (**defragmentation**)

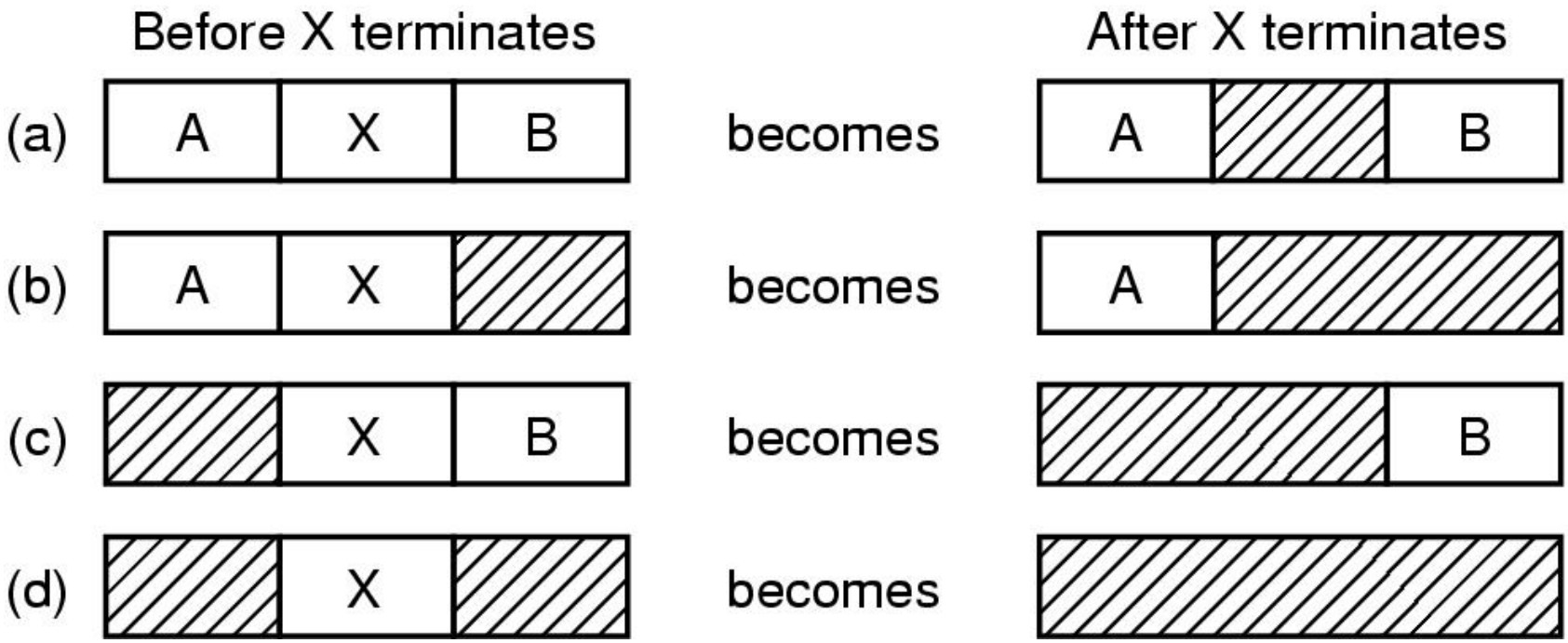


Dynamic partitioning: Memory Management with Bit Maps and linked lists [AT MOS 2e]



- Part of memory with 5 occupied segments, 3 holes
  - tick marks show allocation units
  - shaded regions are free
- B - Corresponding bit map
- C - Same information as a list

Memory Management with Linked Lists: need of merge operations (AT MOS 2e-book)



Four neighbor combinations for the terminating process/segment X

Can also use more advanced data structures; cf. Buddy systems ch 9 SGG-book (we do not study this closer)

Dynamic Partitioning: Placement algorithms

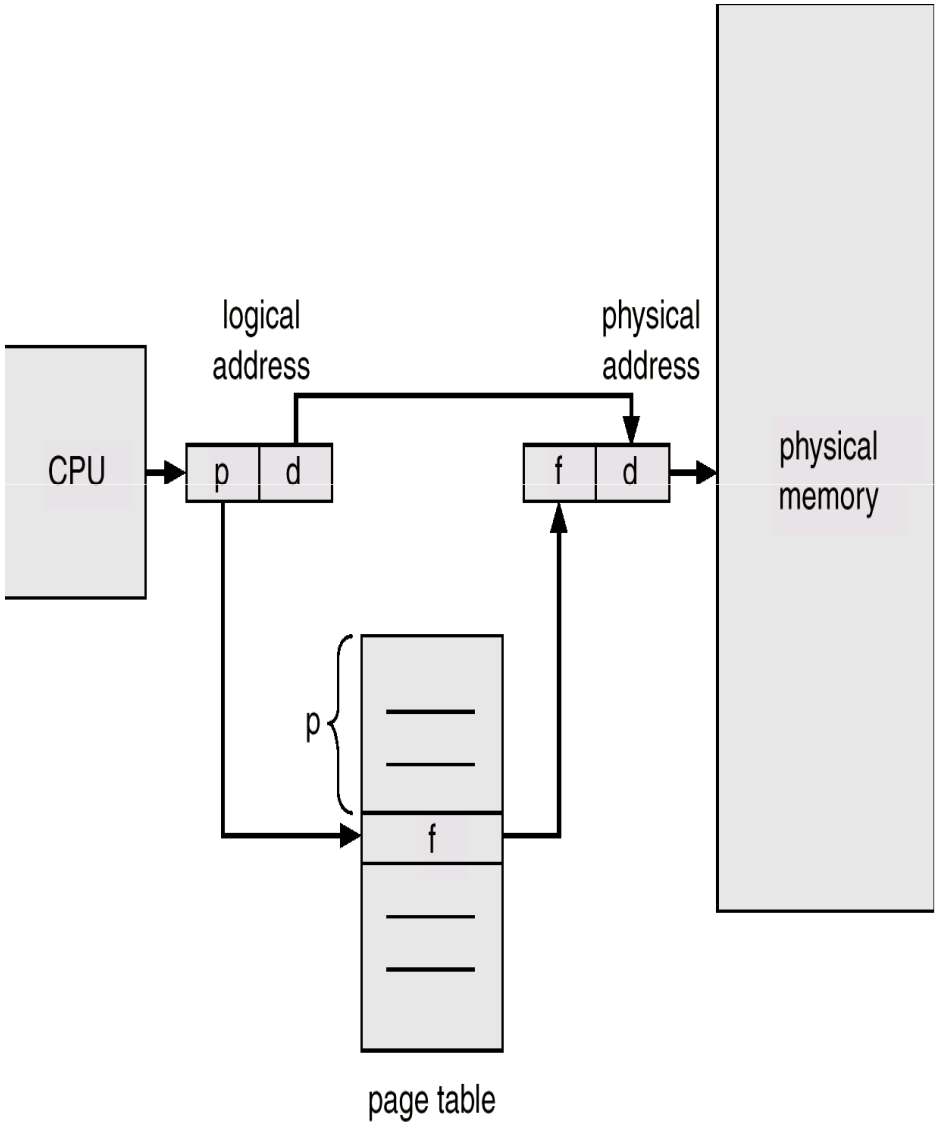
Which available partition to allocate for a request?

- **First-fit**: use the *first* block that is big enough
  - fast
- **Next-fit**: use the *next* block that is big enough
  - tends to eat-up the large block at the end of the memory
- **Best-fit**: use the *smallest* block that is big enough
  - must search entire list (unless free blocks are ordered by size)
  - produces the smallest leftover hole.
- **Worst-fit**: use the *largest* block
  - must also search entire list
  - produces the largest leftover hole…
  - … but eats-up big blocks

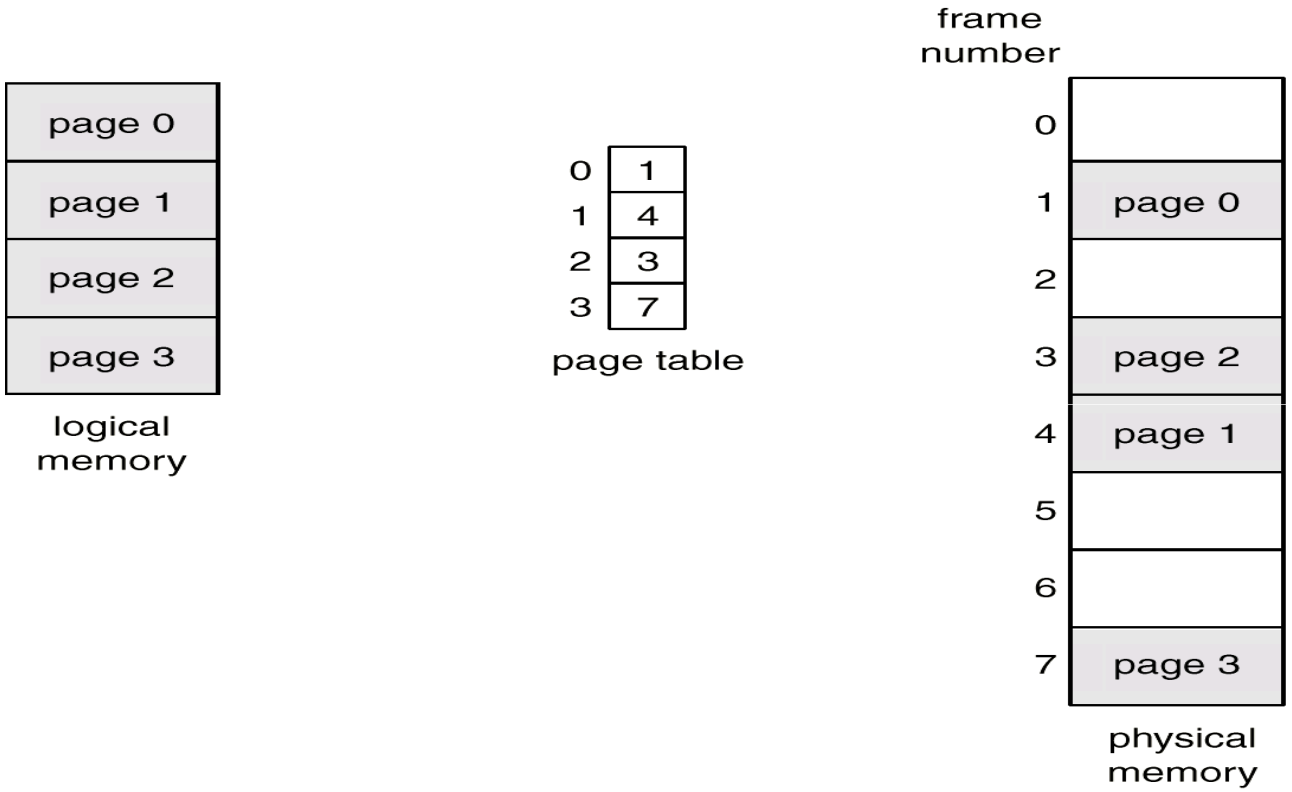
Q: how to avoid external fragmentation?

To avoid external fragmentation: Paging

- Partition memory into small equal-size chunks (**frames**) and divide each process into the same size chunks (**pages**)
- OS maintains a **page table** for each process
  - contains the frame location for each page in the process
  - memory address = (page number, offset within page)

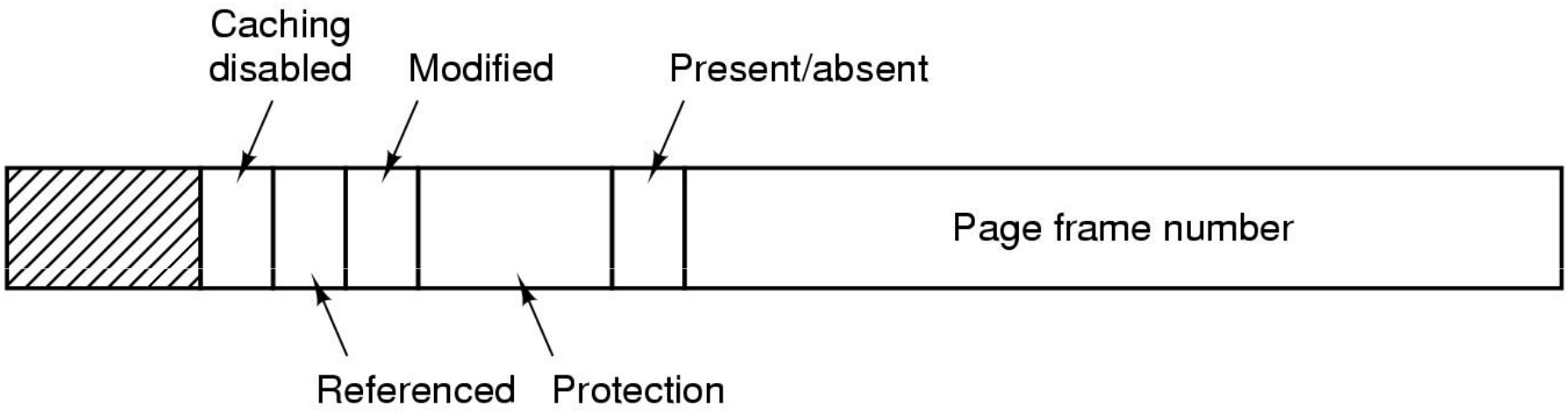


Paging Example



Question: do we avoid fragmentation completely?

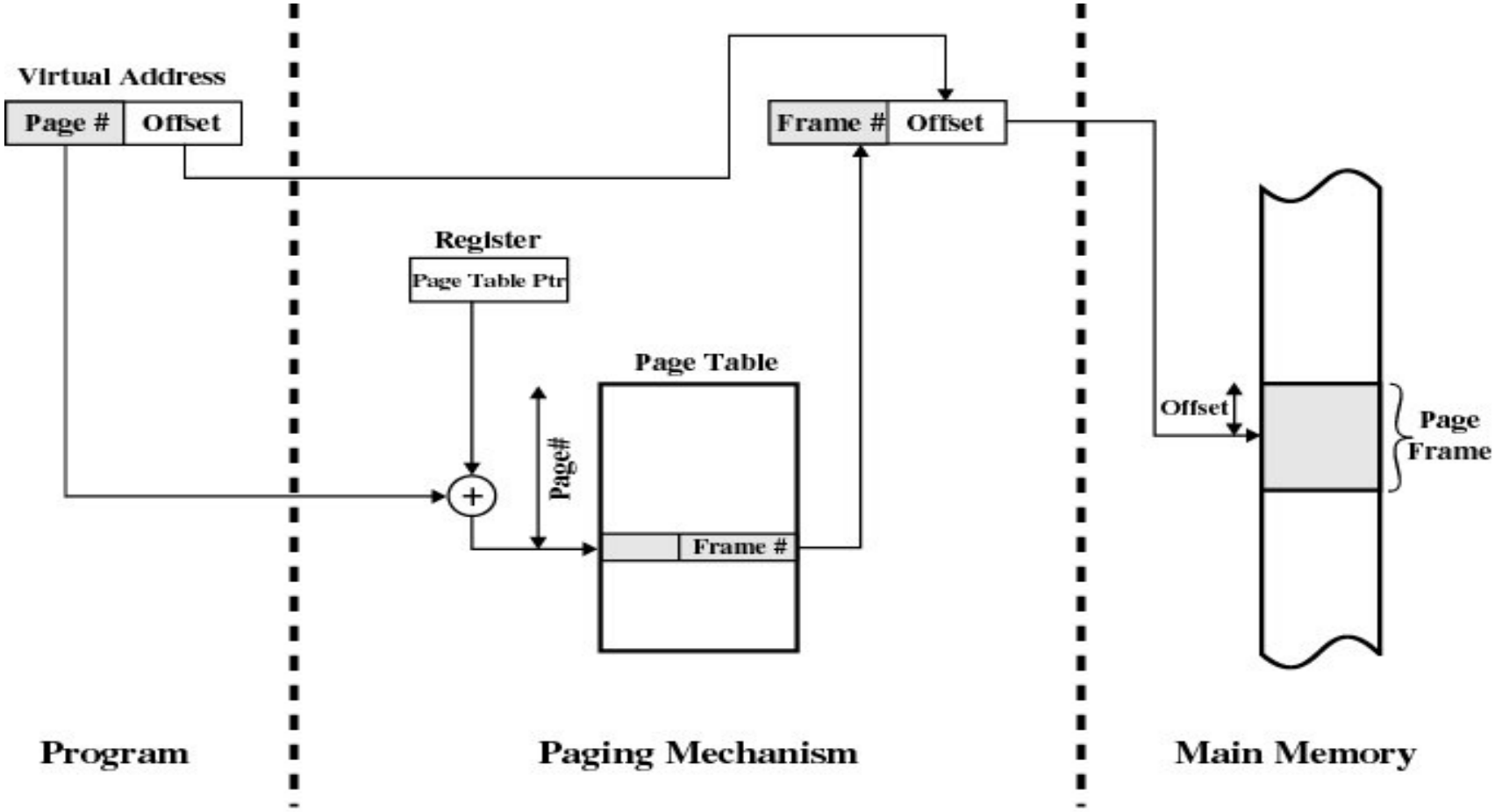
Typical page table entry



(Fig. From A. Tanenbaum, Modern OS 2/e)

Implementation of Page Table?
1. Main memory:

- *page-table base, length registers*
- each program reference to memory => 2 memory accesses



Implementation of Page Table? 2: Associative Registers

a.k.a Translation Lookaside Buffers (TLBs): special fast-lookup hardware cache; parallel search (cache for page table)

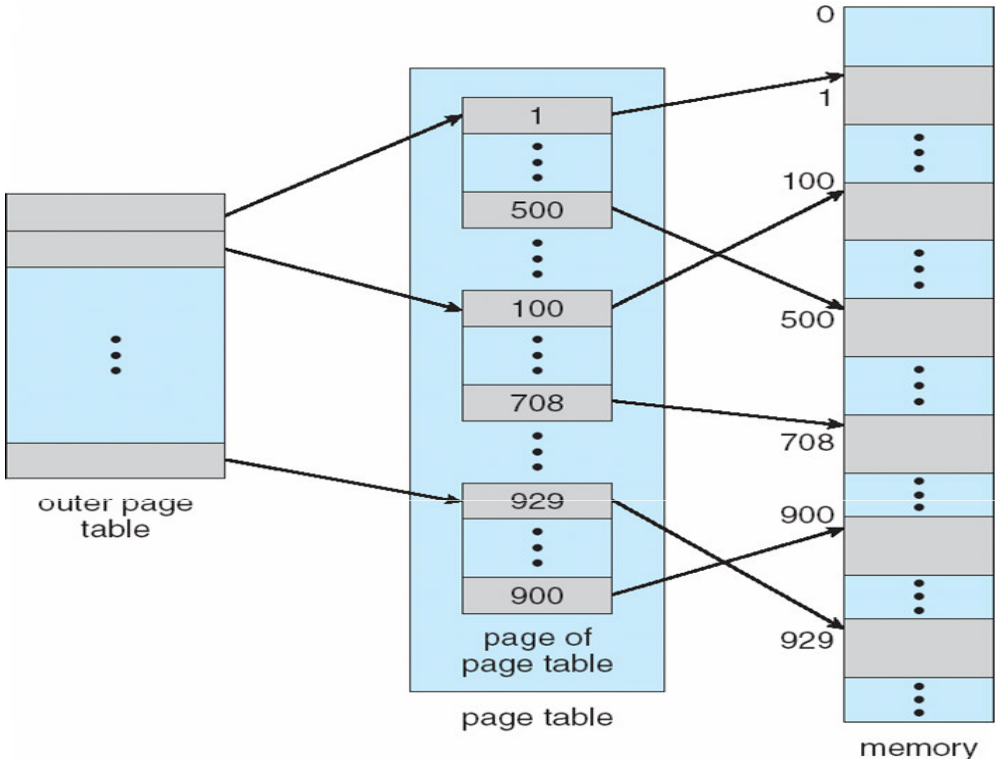
Address translation (P, O): if P is in associative register (**hit**), get frame # from TLB; else get frame # from page table in memory

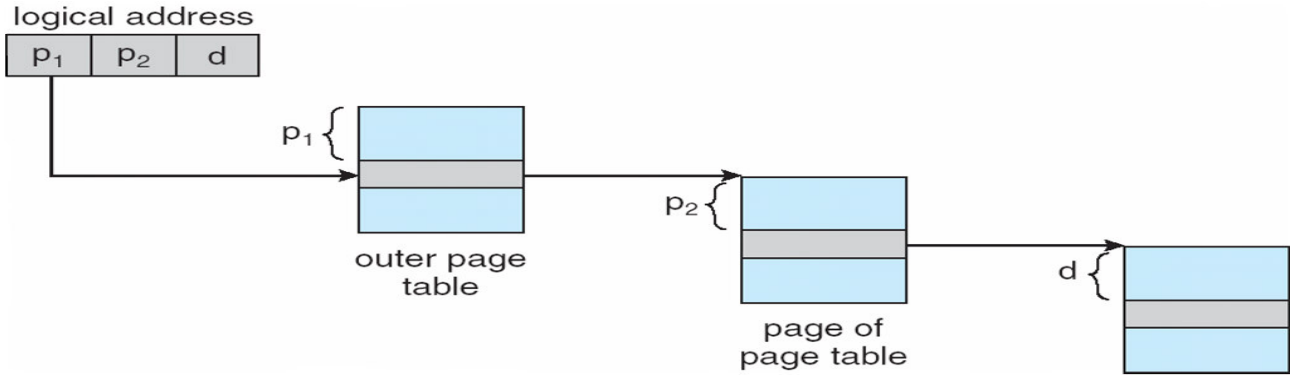
Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

Effective Access Time

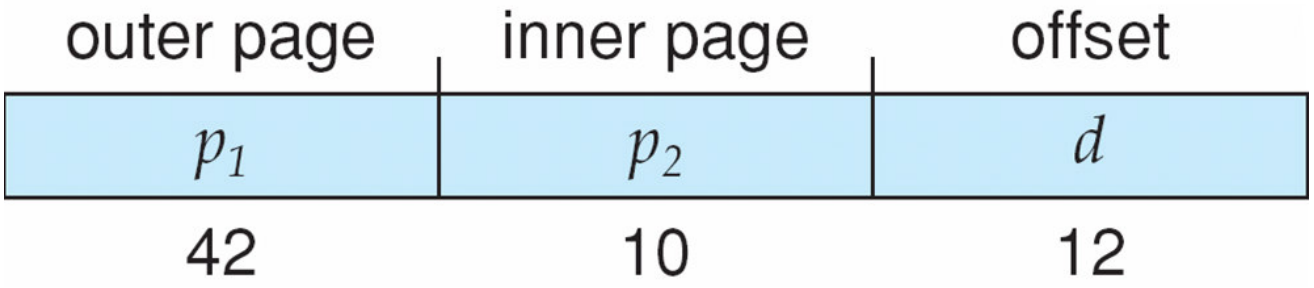
- Associative Lookup = ε time units (fraction of microsecond)
- Assume memory cycle time is 1 microsecond
- **Hit ratio** (= α): percentage of times a page number is found in the associative registers
- Effective Access Time = $(1 + \varepsilon)\,\alpha + (2 + \varepsilon)(1 - \alpha) = 2 + \varepsilon - \alpha$

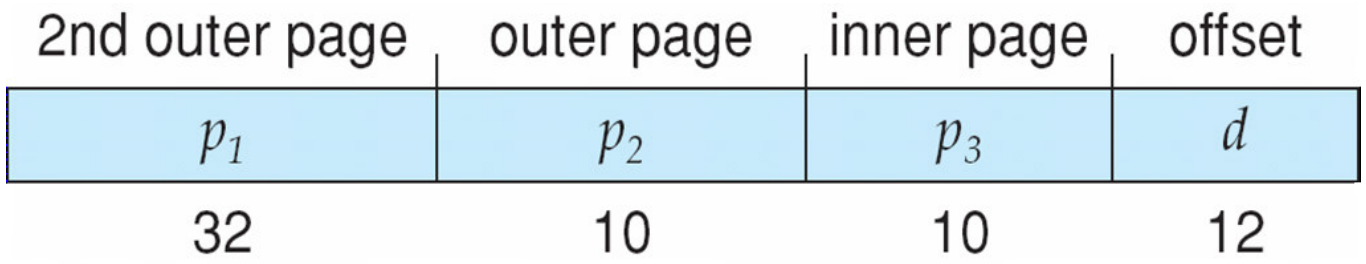
Two-Level Page-Table Scheme and address translation



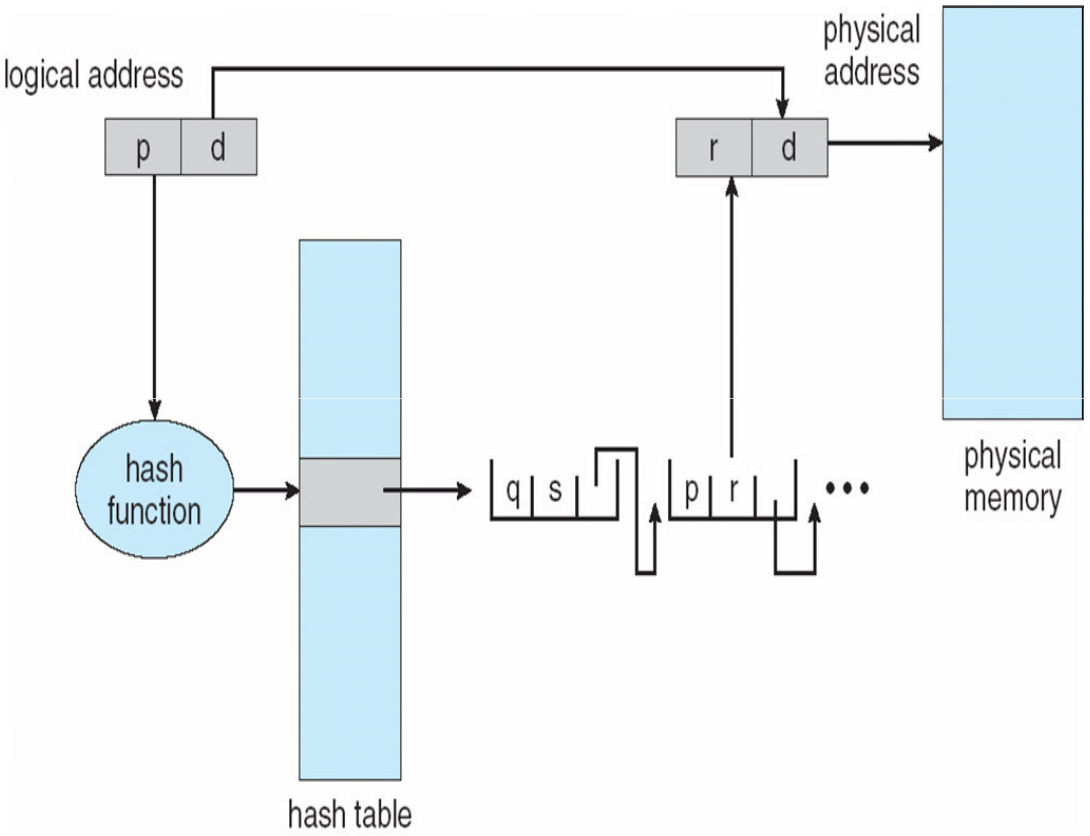


Three-level Paging Scheme



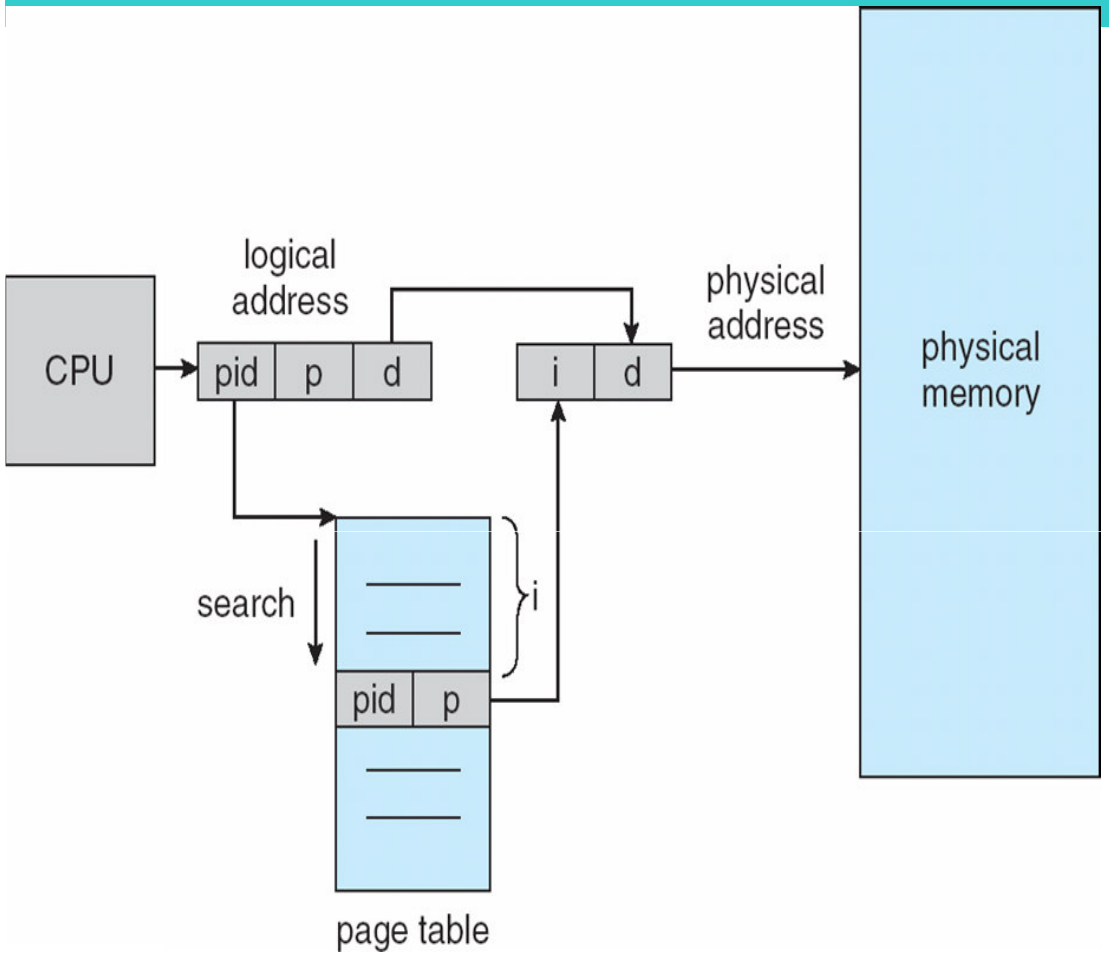


Hashed Page Tables



- 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 number (r in the example) is extracted

Inverted Page Table

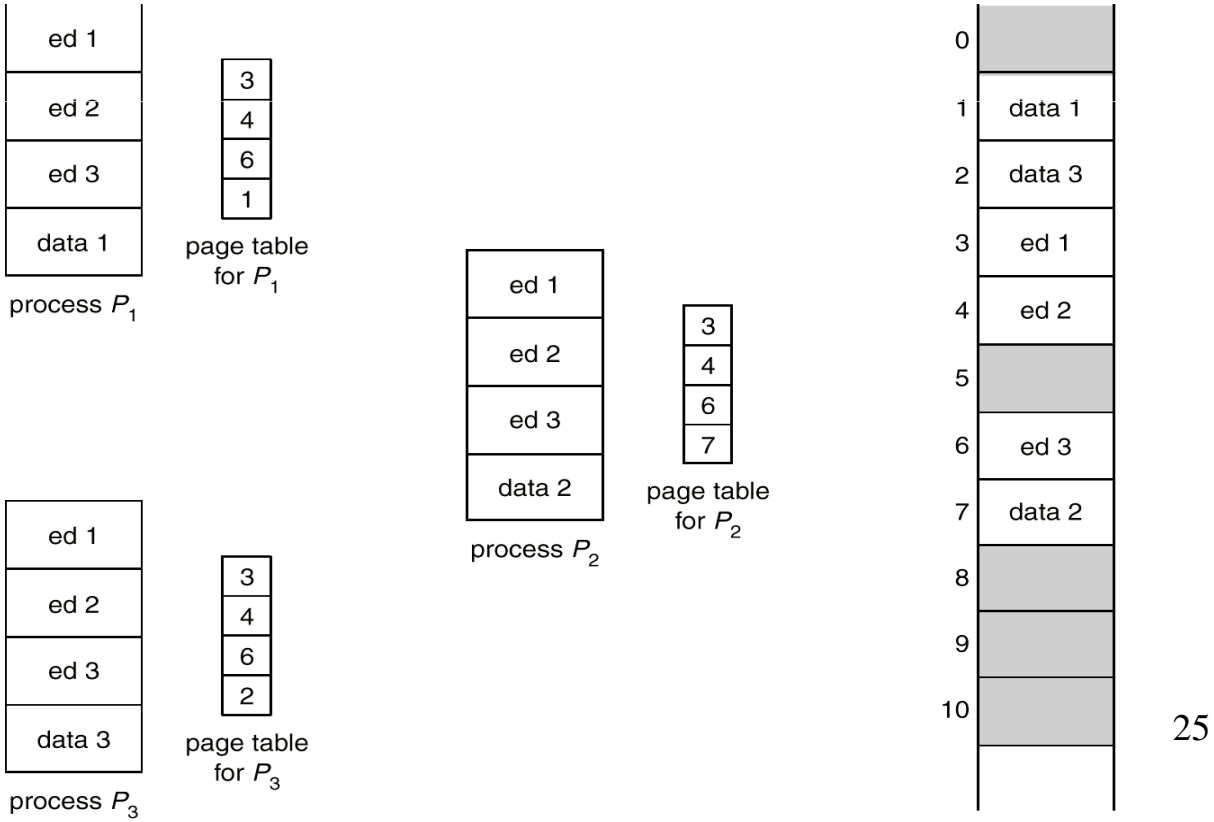


- One entry for each real page of memory
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
- Use hash table to limit the search to one — or at most a few — page-table entries

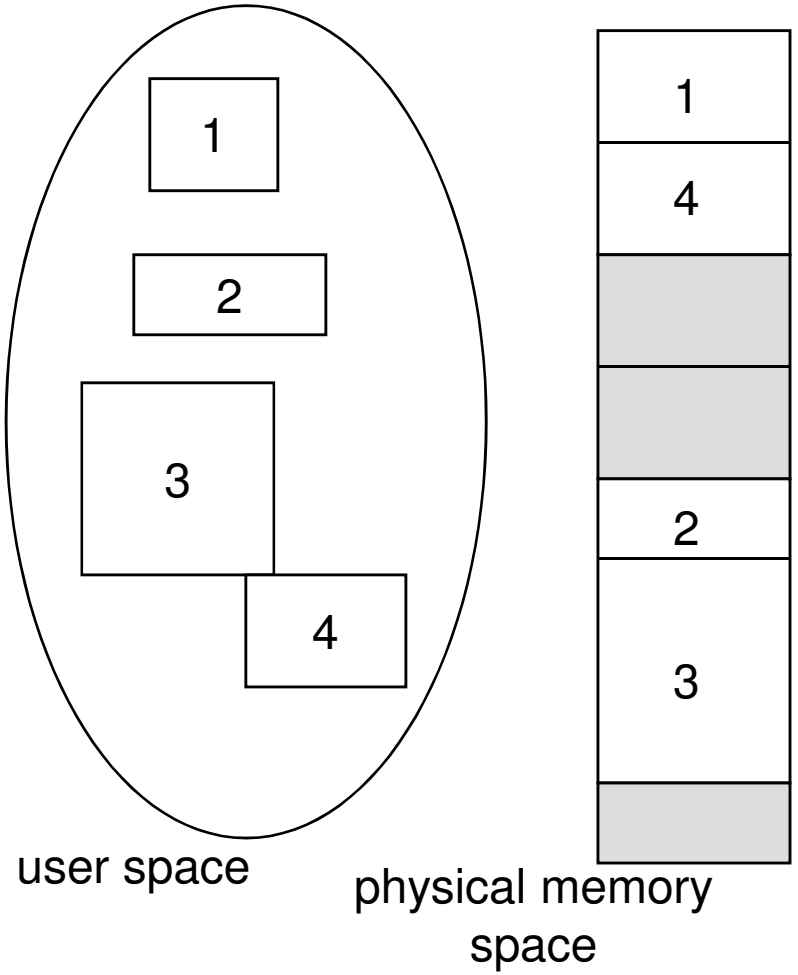
Shared Pages

Shared code: one copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems, library-code, ...).

How to self-address a shared page?: watch for different numbering of page, though; or use indirect referencing



Segmentation

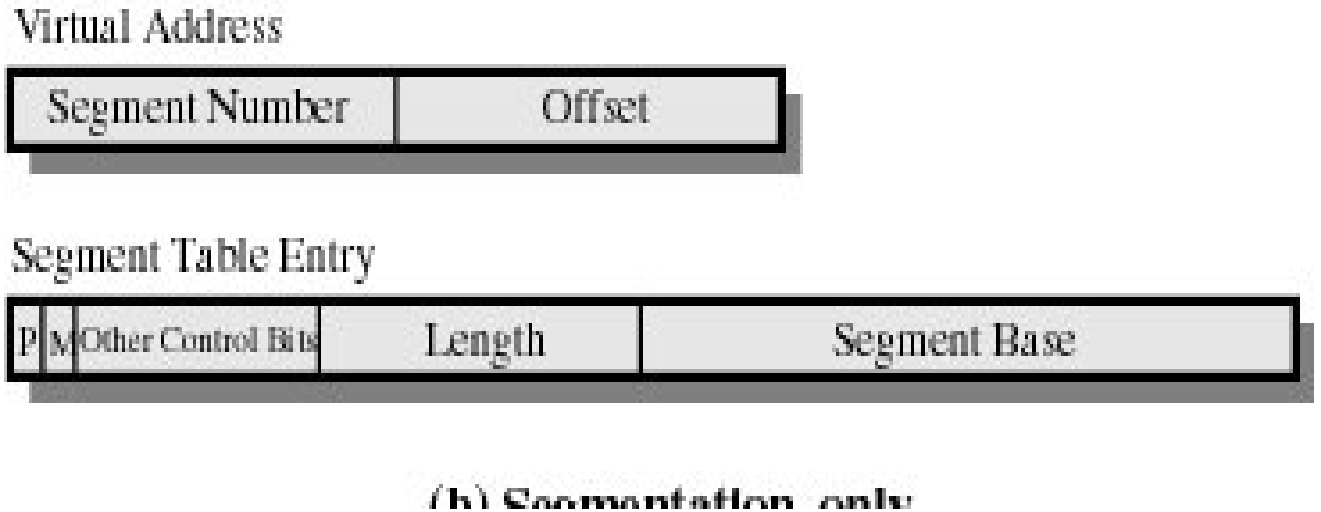


- Memory-management scheme that supports user view of memory/program, i.e. a collection of segments.
- segment = logical unit such as:

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

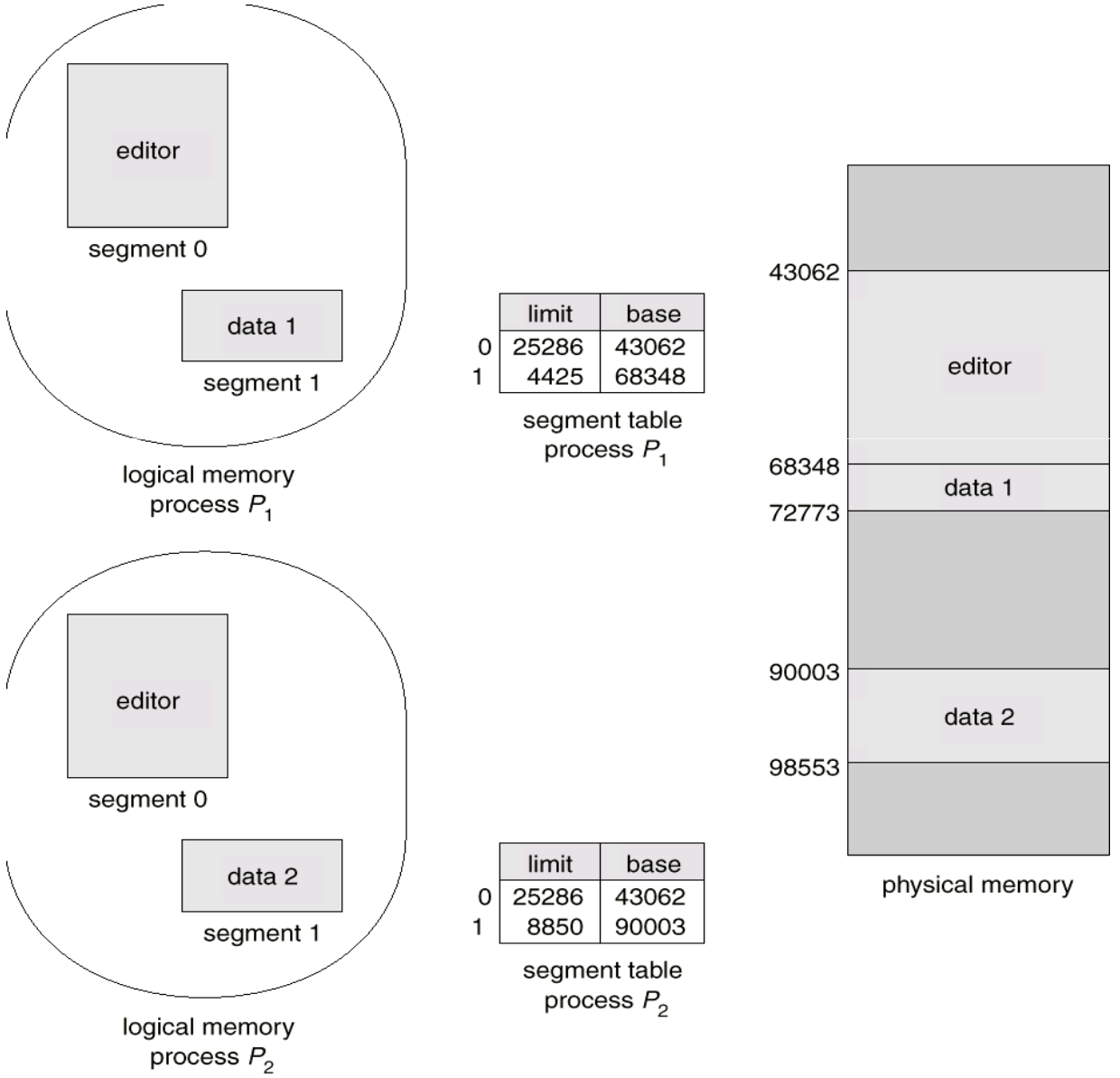
Segmentation Architecture

- Protection: each entry in segment table:
  - validation bit = 0 ⇒ illegal segment
  - read/write/execute privileges
  - ...
- Code sharing at segment level (watch for segment numbers, though; or use indirect referencing).
- Segments vary in length => need dynamic partitioning for memory allocation.

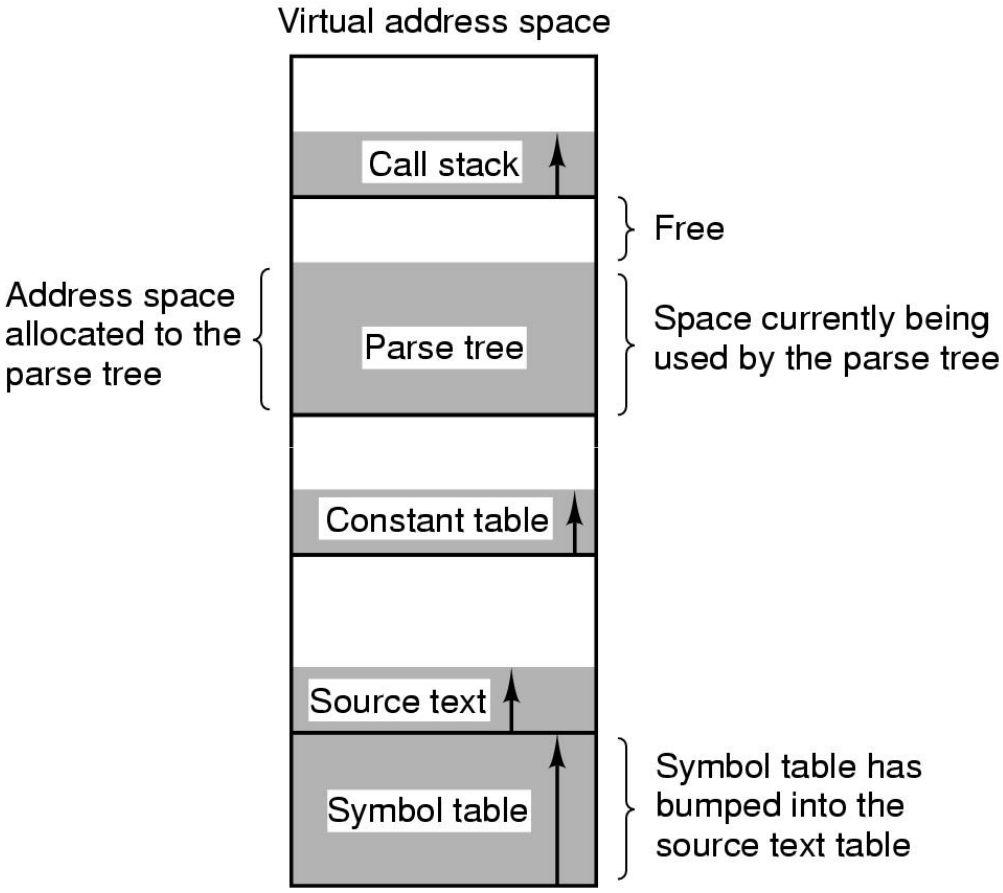


Sharing of segments

Simpler to self-address a shared segment by using the same seg#



Segmentation (A.T. MOS 2/e)



- One-dimensional address space with growing tables
- One table may bump into another

Comparison of paging and segmentation

(A.T. MOS 2/e)

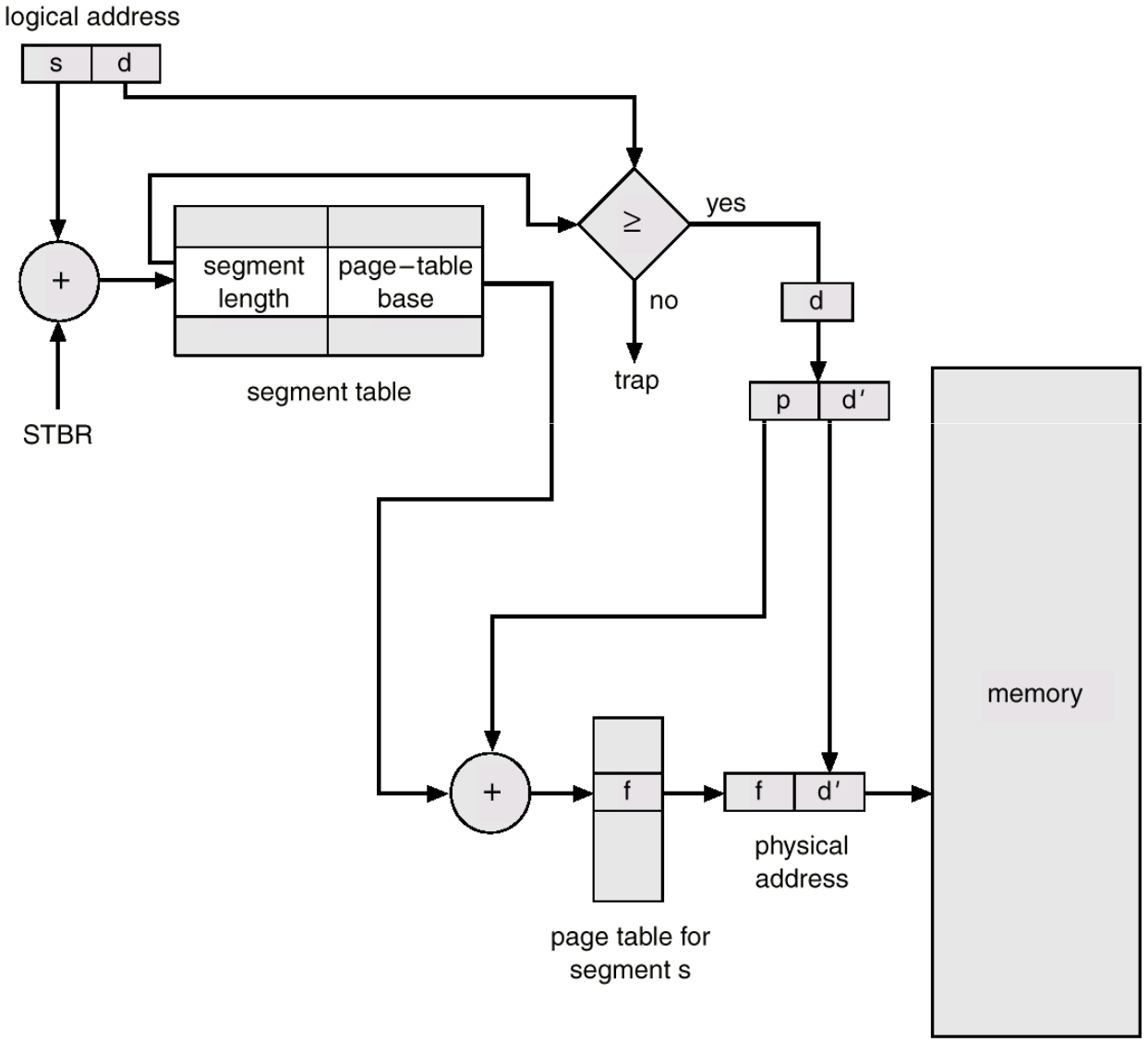
Consideration	Paging	Segmentation
Need the programmer be aware that this technique is being used?	No	Yes
How many linear address spaces are there?	1	Many
Can the total address space exceed the size of physical memory?	Yes	Yes
Can procedures and data be distinguished and separately protected?	No	Yes
Can tables whose size fluctuates be accommodated easily?	No	Yes
Is sharing of procedures between users facilitated?	No	Yes
Why was this technique invented?	To get a large linear address space without having to buy more physical memory	To allow programs and data to be broken up into logically independent address spaces and to aid sharing and protection

Combined Paging and Segmentation

- Paging
  - transparent to the programmer
  - eliminates external fragmentation
- Segmentation
  - visible to the programmer
  - allows for growing data structures, modularity, support for sharing and protection
  - But: memory allocation?

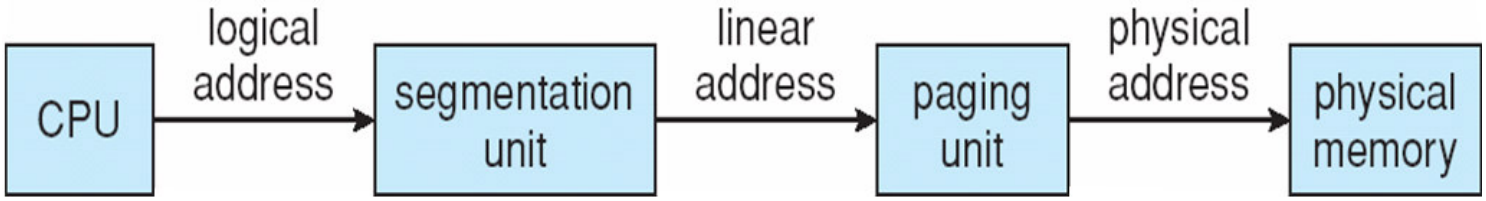
- Hybrid solution: page the segments (each segment is broken into fixed-size pages)
  - E.g. MULTICS, Pentium

Combined Address Translation Scheme

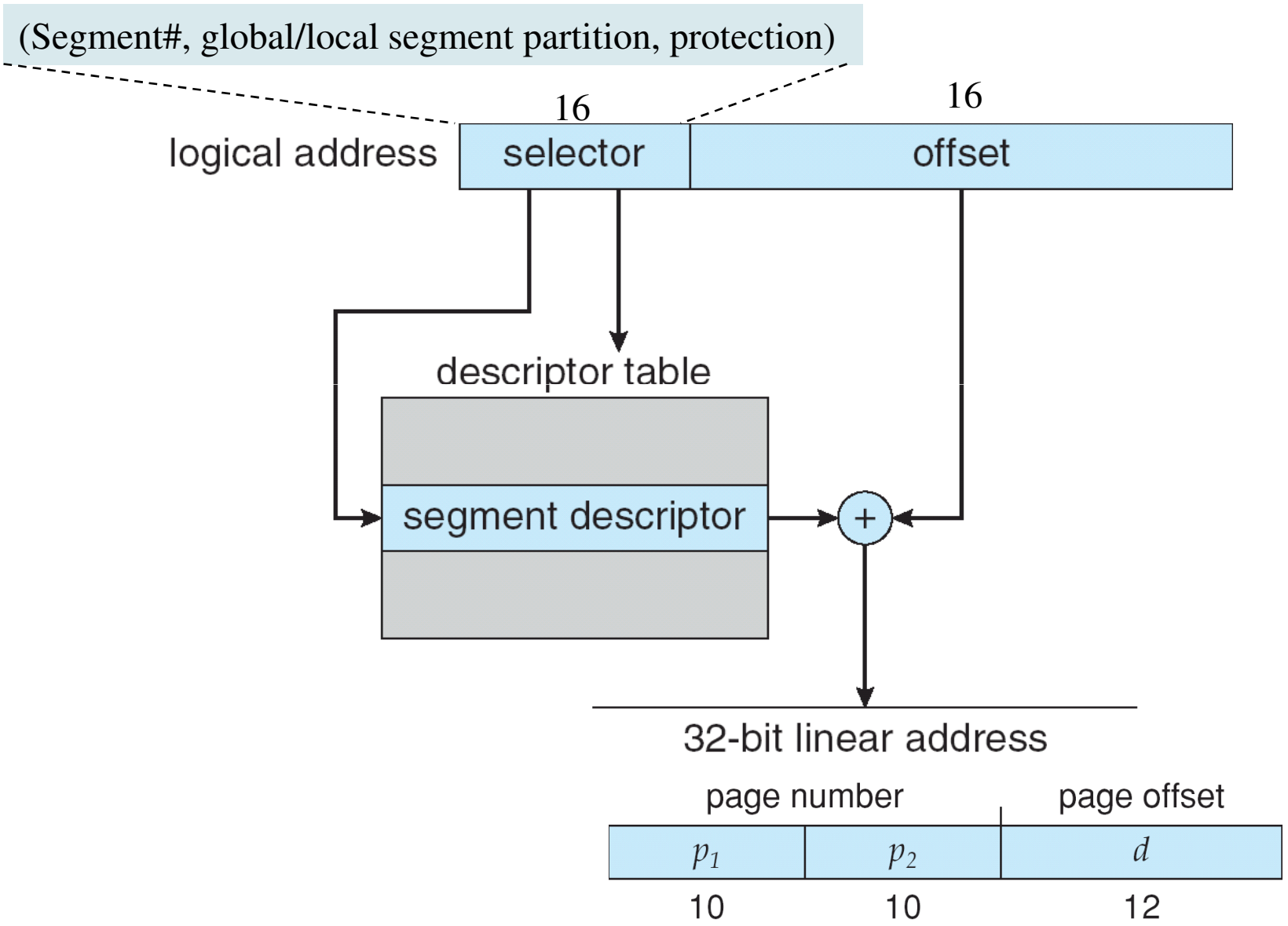


Example: The Intel Pentium

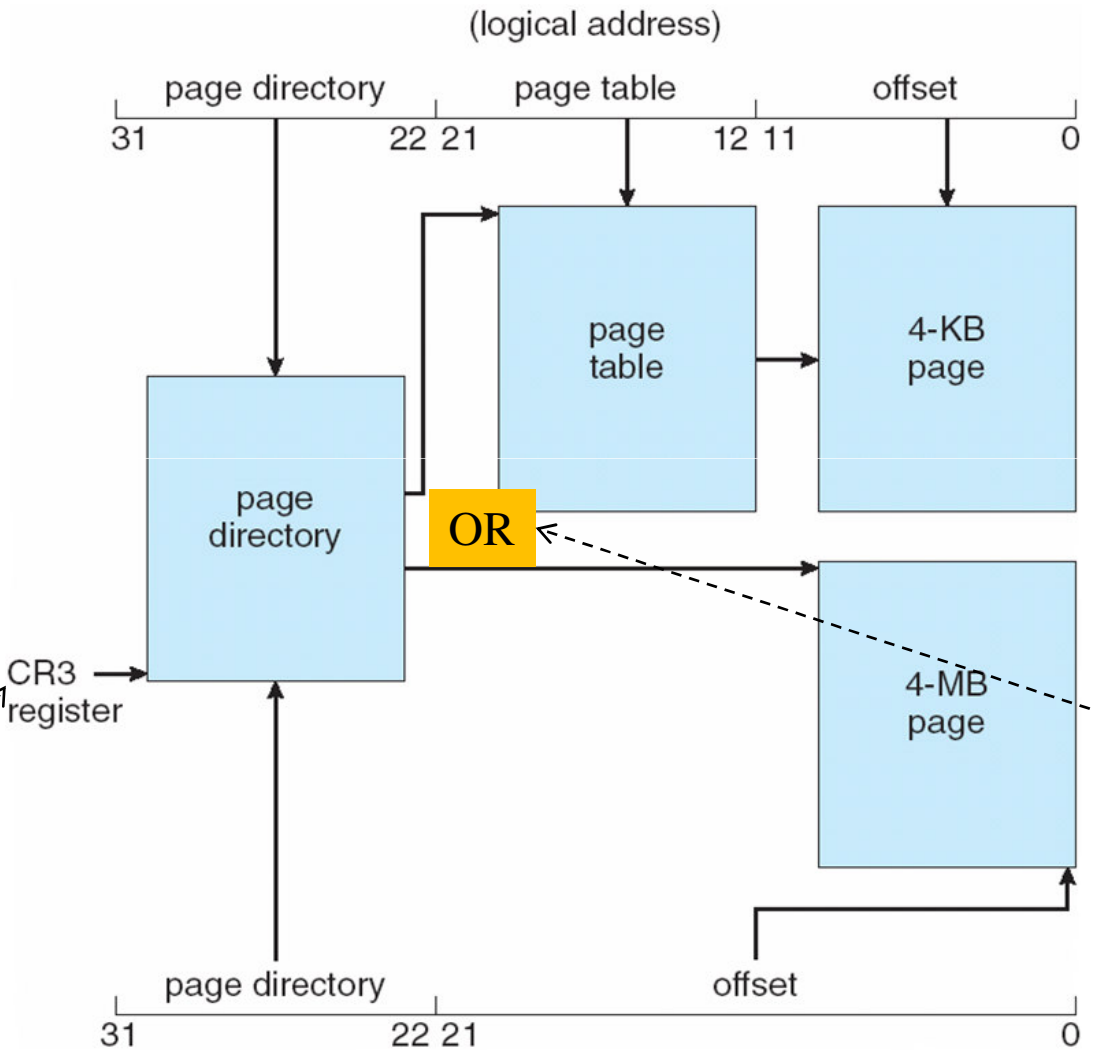
- Supports both segmentation and segmentation with paging
- CPU generates logical address
  - Given to segmentation unit
    - Which produces linear addresses
  - Linear address given to paging unit
    - Which generates physical address in main memory
    - Paging units form equivalent of MMU



Intel Pentium Segmentation



Pentium Paging Architecture



One of the processor's registers

Depending on flag in page-directory

Linear Address in Linux in Pentium Architecture

Supports fixed # of segments

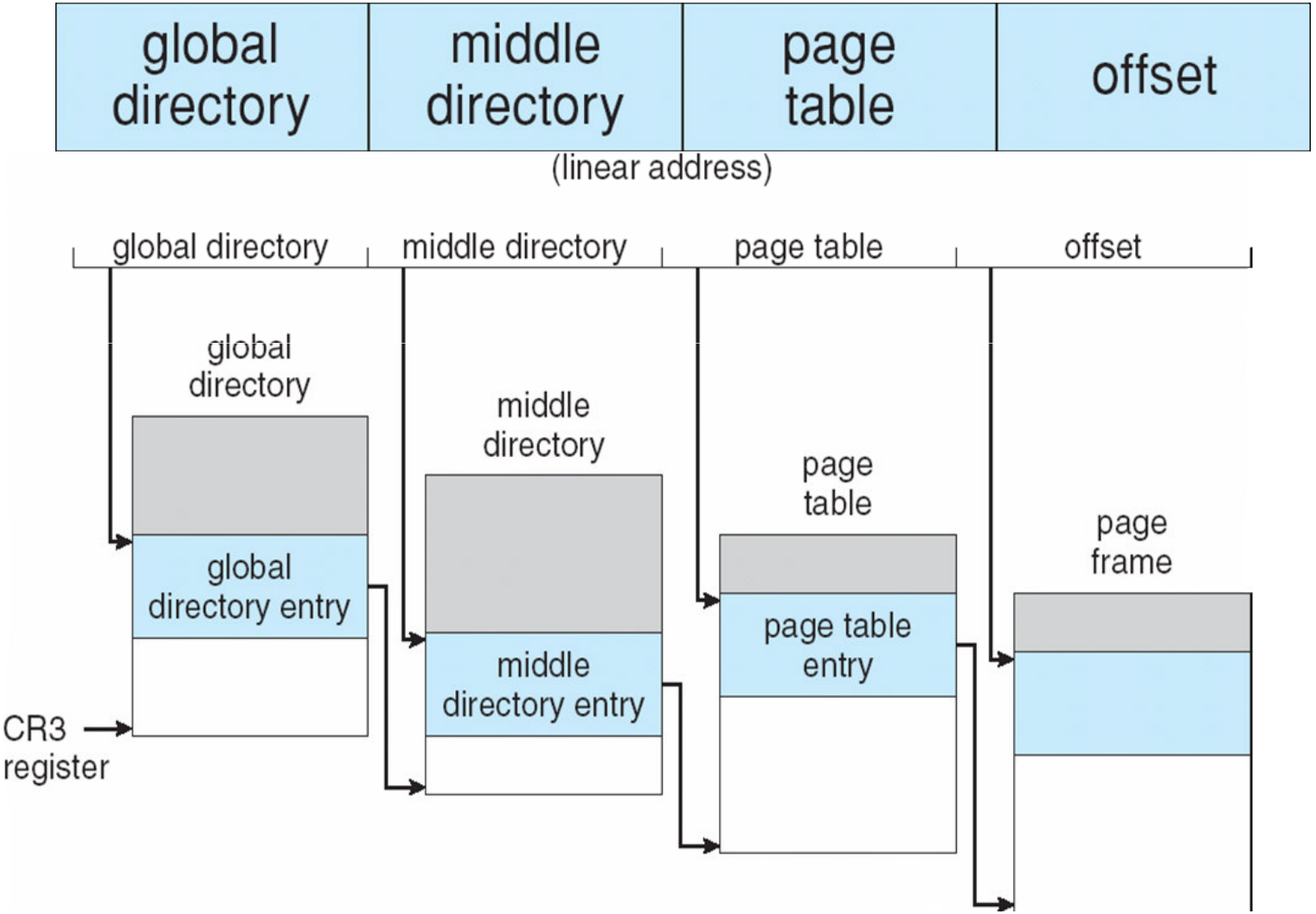
- for portability (not all architectures support segmentation)
- kernel code/data, user code/data, task-state segment (data useful for context switching), local data segment (usually some default)

Needs to comply with 32 and 64-bit architectures

- Uses 3 level-paging (see next)

Three-level Paging in Linux

0-bits in32-bit pendium



Segmentation with Paging: MULTICS
(A.T. MOS 2/e)

Comparison field: Segment number	Comparison field: Virtual page	Page frame	Protection	Age	Is this entry used?
4	1	7	Read/write	13	1
6	0	2	Read only	10	1
12	3	1	Read/write	2	1
					0
2	1	0	Execute only	7	1
2	2	12	Execute only	9	1

- Simplified version of the MULTICS TLB
- Existence of 2 page sizes makes actual TLB more complicated (cf pentium outline)

Virtual memory

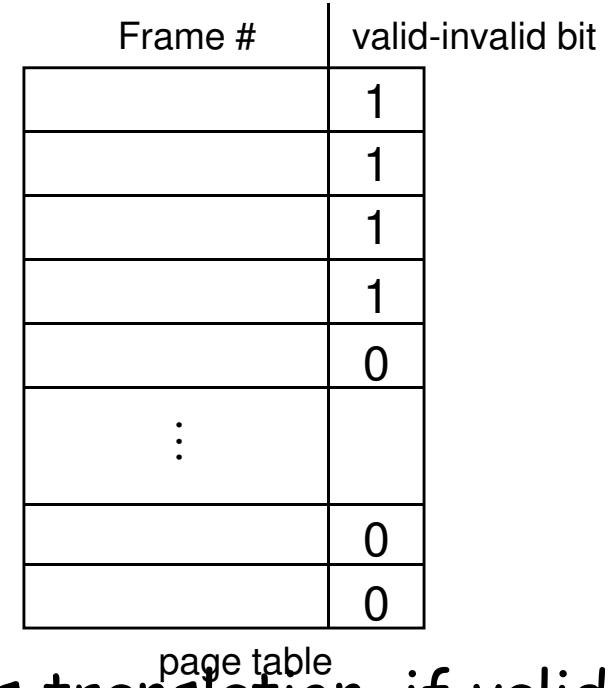
Execution of a program: Virtual memory concept

Main memory = cache of the disk space

- Operating system brings into main memory a few pieces of the program
- Resident set - portion of process that is in main memory
- when an address is needed that is not in main memory a **page-fault** interrupt is generated:
  - OS places the process in blocking state and issues a disk IO request
  - another process is dispatched

Valid-Invalid Bit

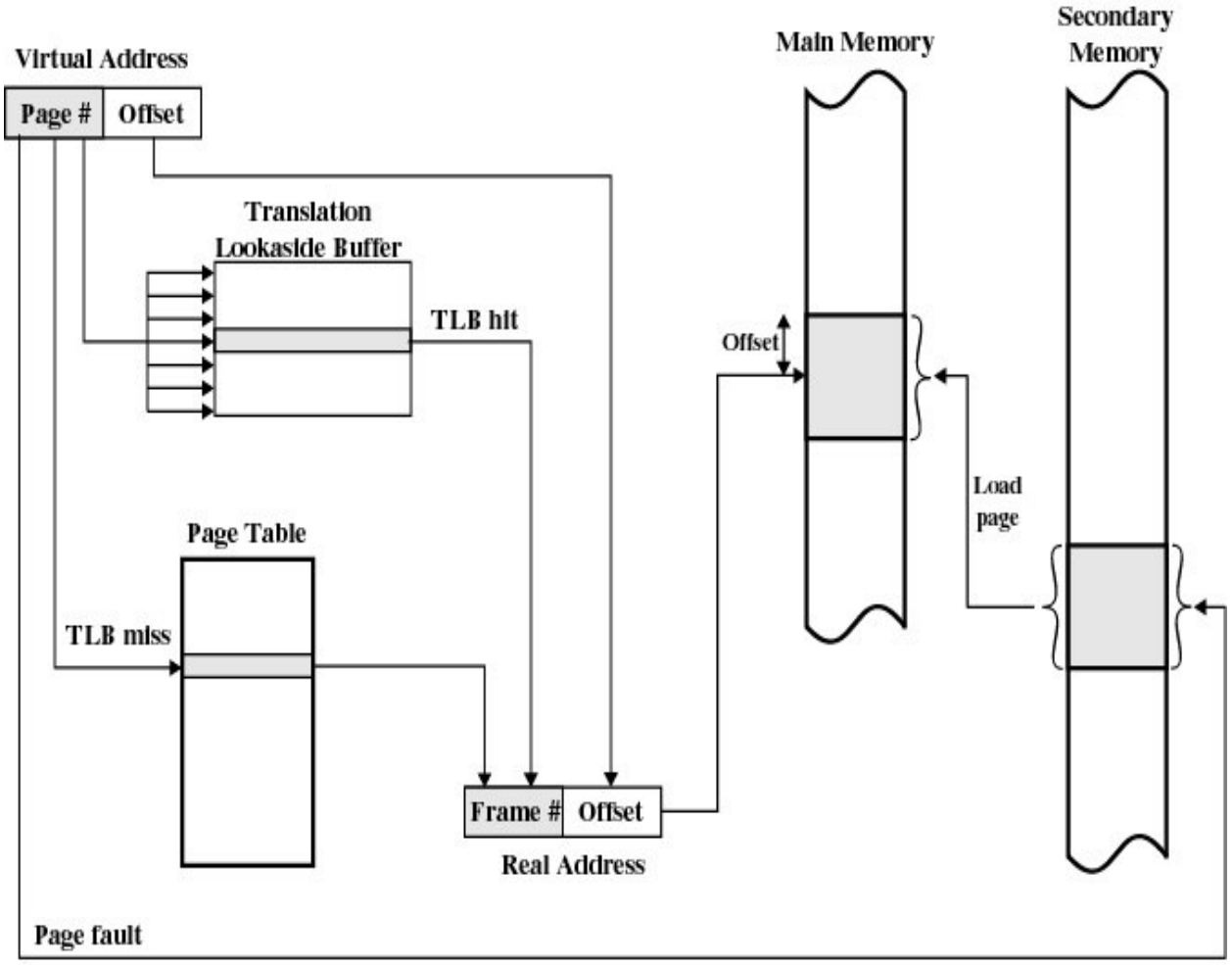
- With each page table entry a valid–invalid bit is associated (initially 0)
  - 1 ⇒ in-memory
  - 0 ⇒ not-in-memory



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

Page Fault and (almost) complete address-translation scheme

In response to page-fault interrupt, OS must:



- get empty frame (swap out that page?).
- swap in page into frame.
- reset tables, validation bit
- restart instruction

Figure 8.7 Use of a Translation Lookaside Buffer

if there is no free frame?

Page replacement –want an algorithm which will result in minimum number of page faults.

- Page fault forces choice
  - which page must be removed
  - make room for incoming page

- Modified page must first be saved
  - unmodified just overwritten(use *dirty bit* to optimize writes to disk)

- Better not to choose an often used page
  - will probably need to be brought back in soon

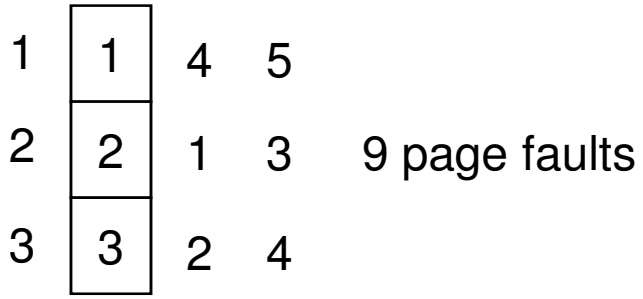
Replacement algorithms in virtual memory

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

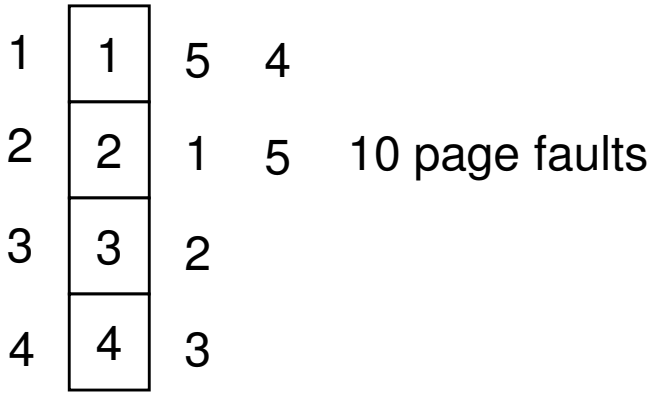
Can be implemented using a circular buffer

Ex.:Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

- 3 frames



- 4 frames



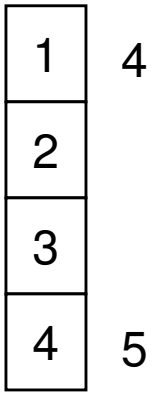
- **Belady's Anomaly**: more frames , sometimes more page faults

  Problem: replaces pages that will be needed soon

Optimal Replacement Algorithm

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

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



6 page faults

- How do we know this info?
  - We don't
- Algo can be used for measuring how well other algorithms perform.

Least Recently Used (LRU) Replacement Algorithm

Idea: Replace the page that has not been referenced for the longest time.

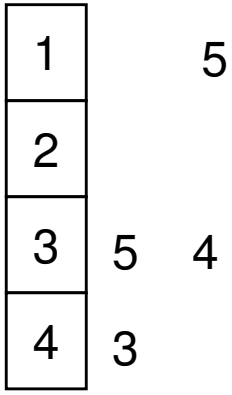
- By the principle of **locality**, this should be the page least likely to be referenced in the near future

Implementation:

- tag each page with the time of last reference
- use a stack

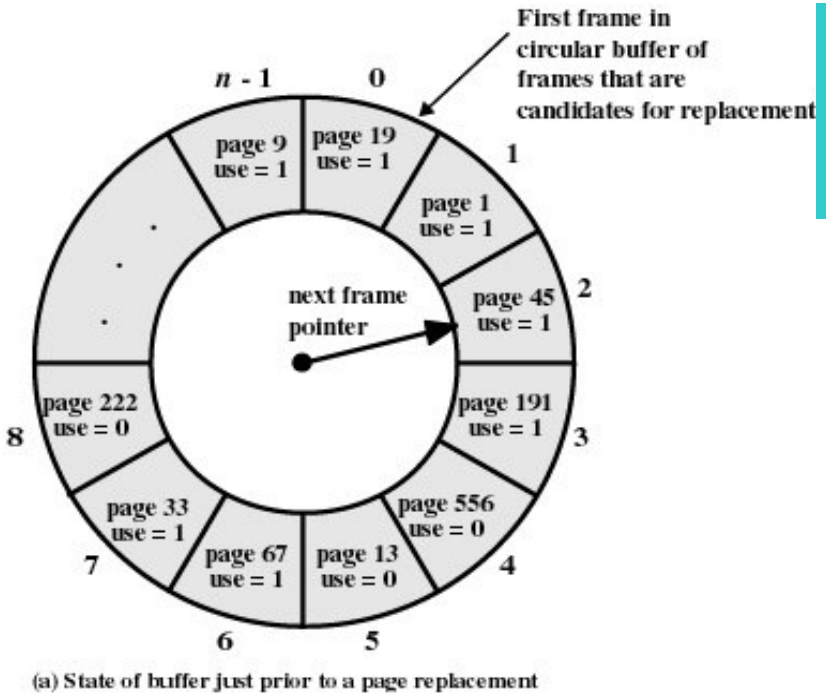
Problem: high overhead (OS kernel involvement at every memory reference!!!) if HW support not available

LRU Algo (cont)



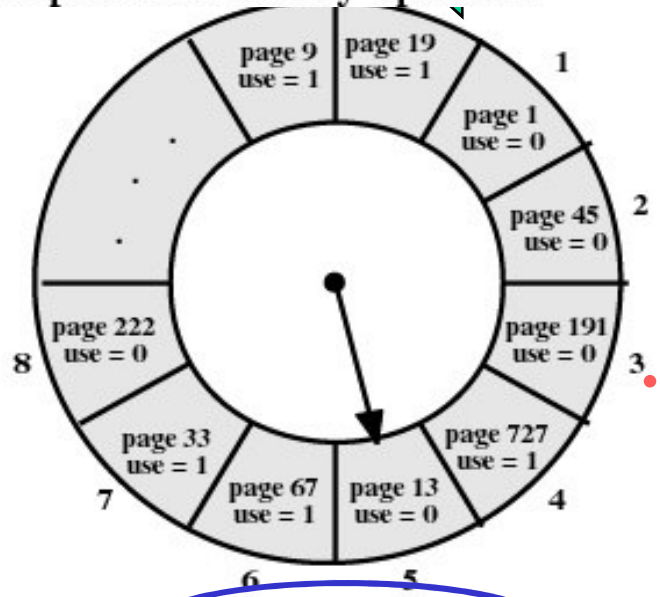
Example: Reference string:

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



(a) State of buffer just prior to a page replacement

Figure 8.16 Example of Clock Policy Operation



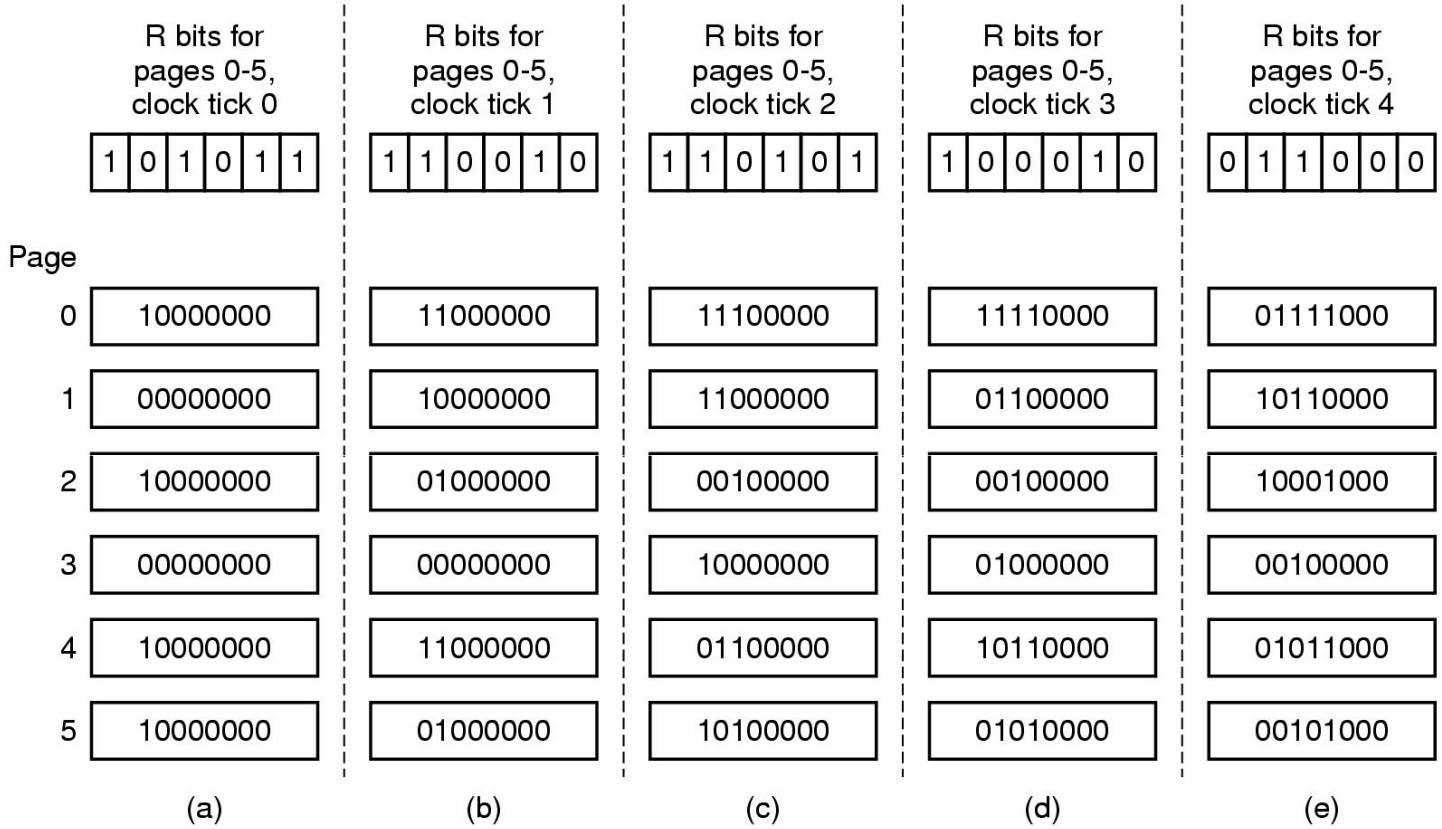
(b) State of buffer just after the next page replacement

LRU Approximations:

Clock/Second Chance -

- uses *use (reference) bit* :
  - initially 0
  - when page is referenced, set to 1 by HW
- to replace a page:
  - the first frame encountered with use bit 0 is replaced.
  - during the search for replacement, each use bit set to 1 is changed to 0 by OS
- note: if all bits set => FIFO

Simulating LRU: the aging algorithm (A.B. MOS 2/e)



- The aging algorithm simulates LRU in software

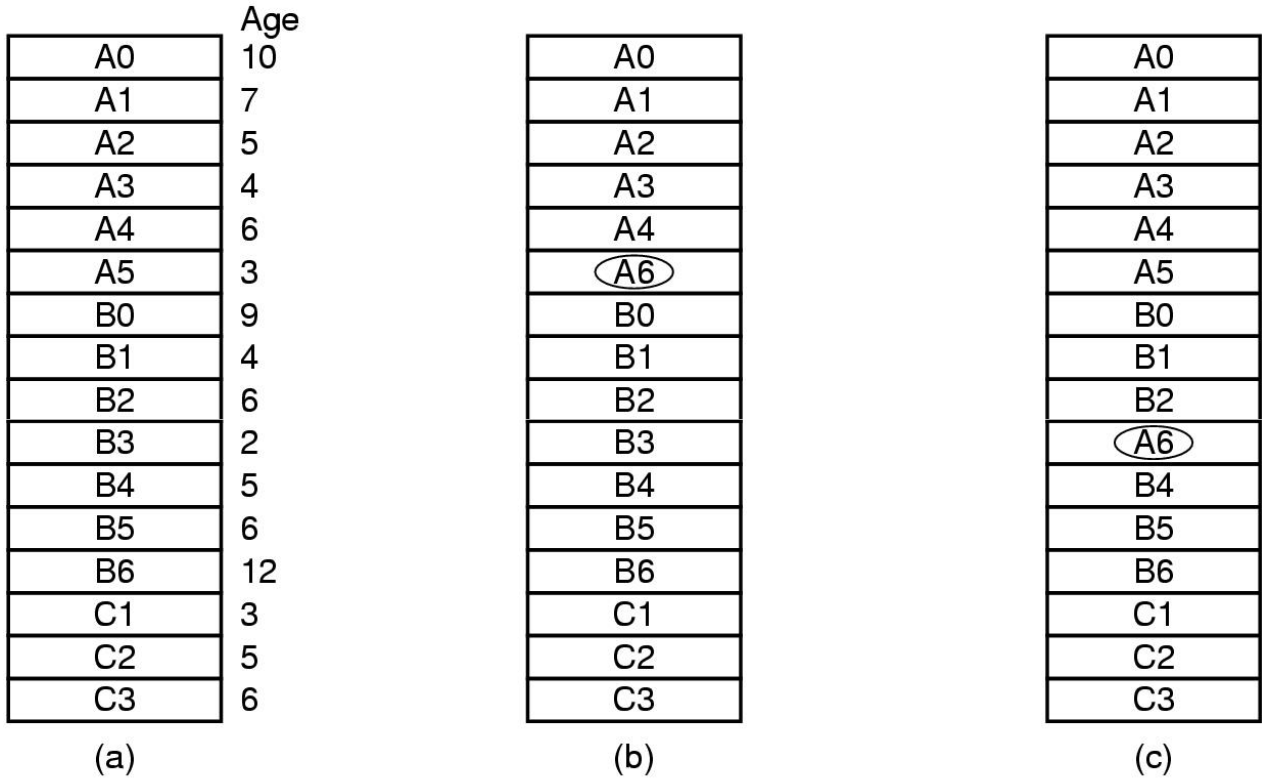
LRU Approximations: Not Recently Used Page Replacement Algorithm

- Each page has Reference (use) bit, Modified (dirty) bit
  - bits are set when page is referenced, modified
  - Ref bit is cleared regularly
- Pages are classified
  1. not referenced, not modified
  2. not referenced, modified (is it possible?)!
  3. referenced, not modified
  4. referenced, modified
- NRU removes page at random from lowest numbered non empty class

Design Issues for Paging Systems

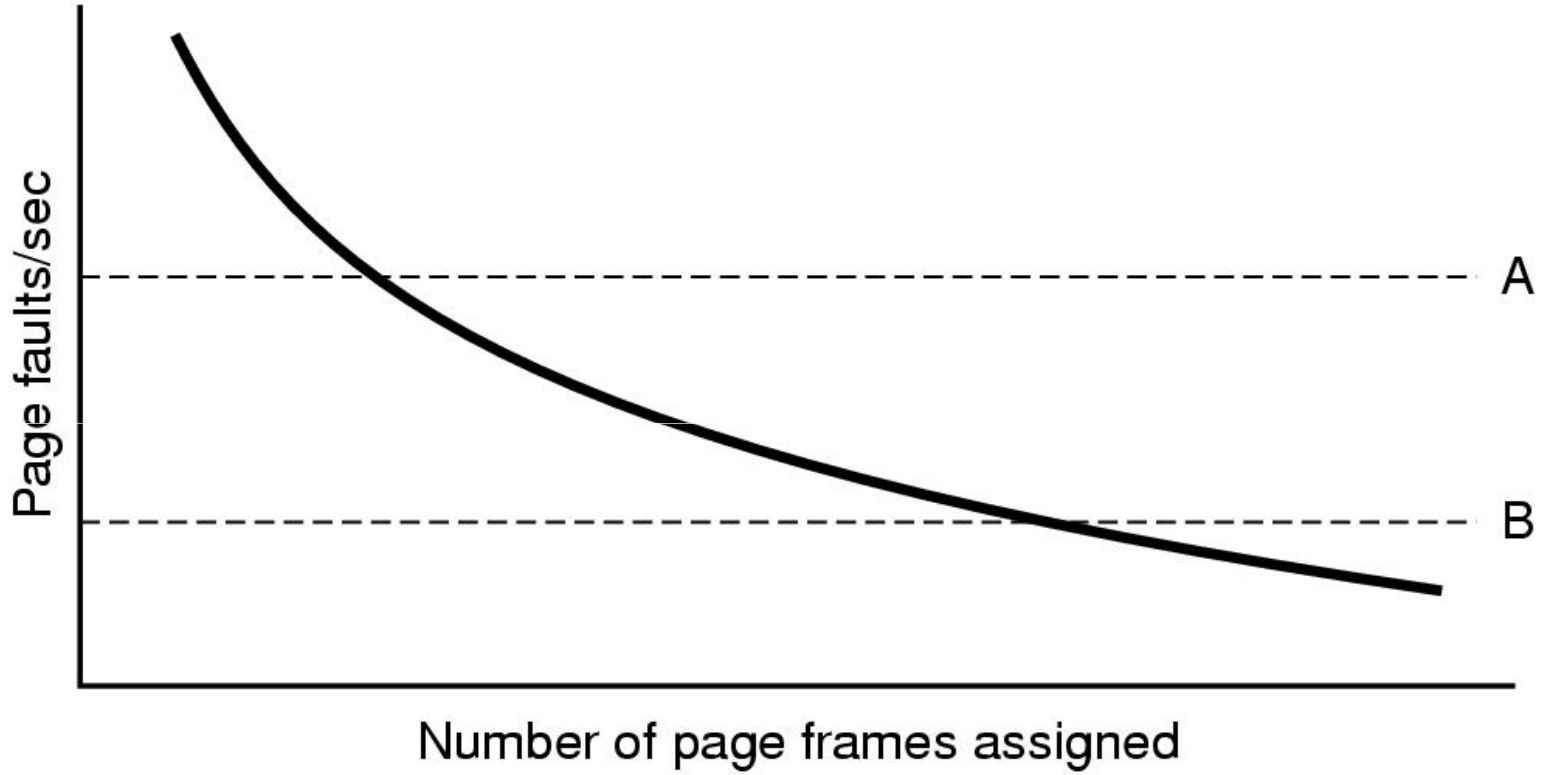
- Global vs local allocation policies
  - Of relevance: Thrashing, working set
- Cleaning Policy
- Fetch Policy
- Page size

Local versus Global Allocation Policies (A.T. MOS2/e)



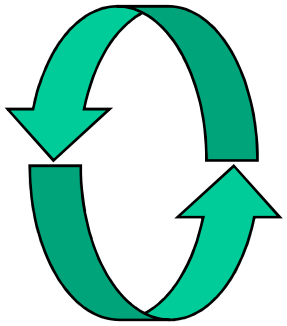
- Original configuration
- Local page replacement
- Global page replacement

Local versus Global Allocation Policies (A.T. MOS2/e)



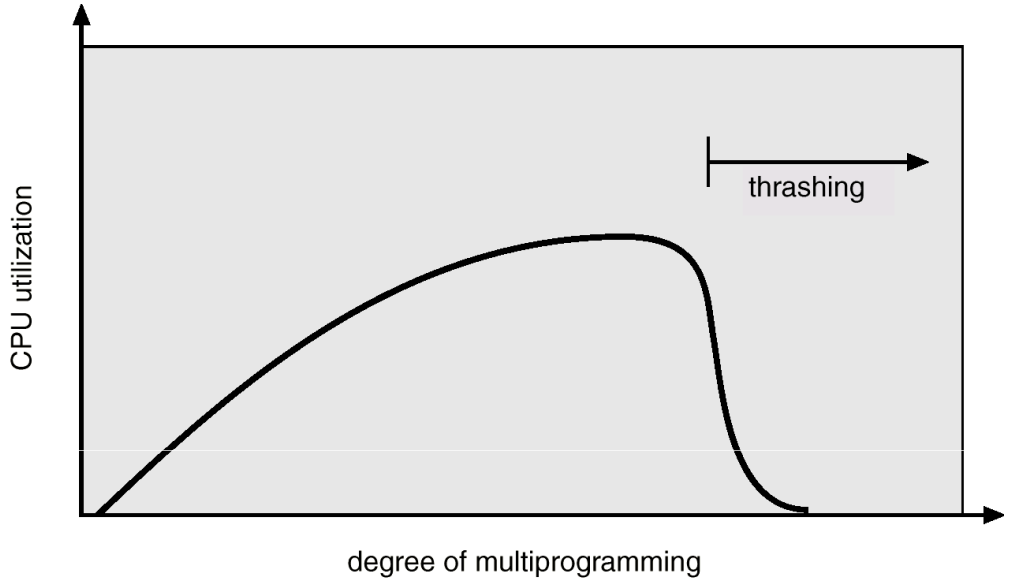
Per-process page fault rate as a function of the number of page frames assigned to the process

Thrashing



- If a process does not have "enough" pages, the page-fault rate is very high. This leads to:
  - low CPU utilization.
  - operating system may think that it needs to increase the degree of multiprogramming.
  - another process added to the system…
  - and the cycle continues …
- Thrashing ≡ the system is busy serving page faults (swapping pages in and out).

Thrashing Diagram



Why does paging work?

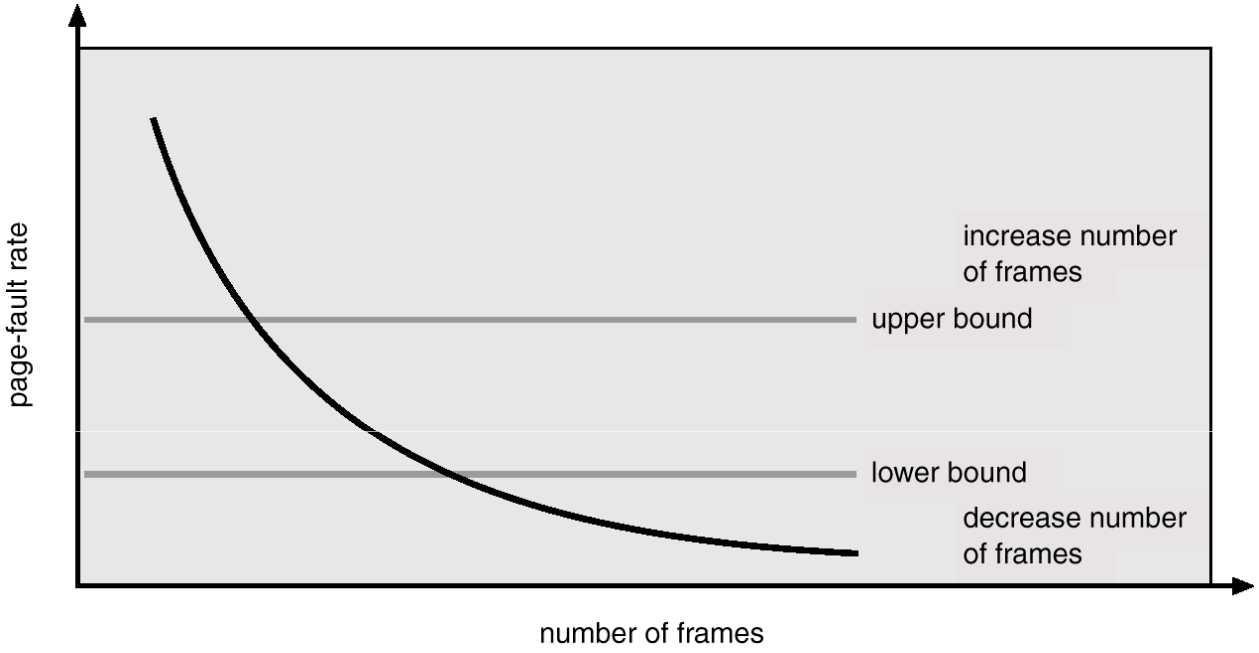
Locality model

- Process migrates from one locality to another.
- Localities may overlap.

Why does thrashing occur?

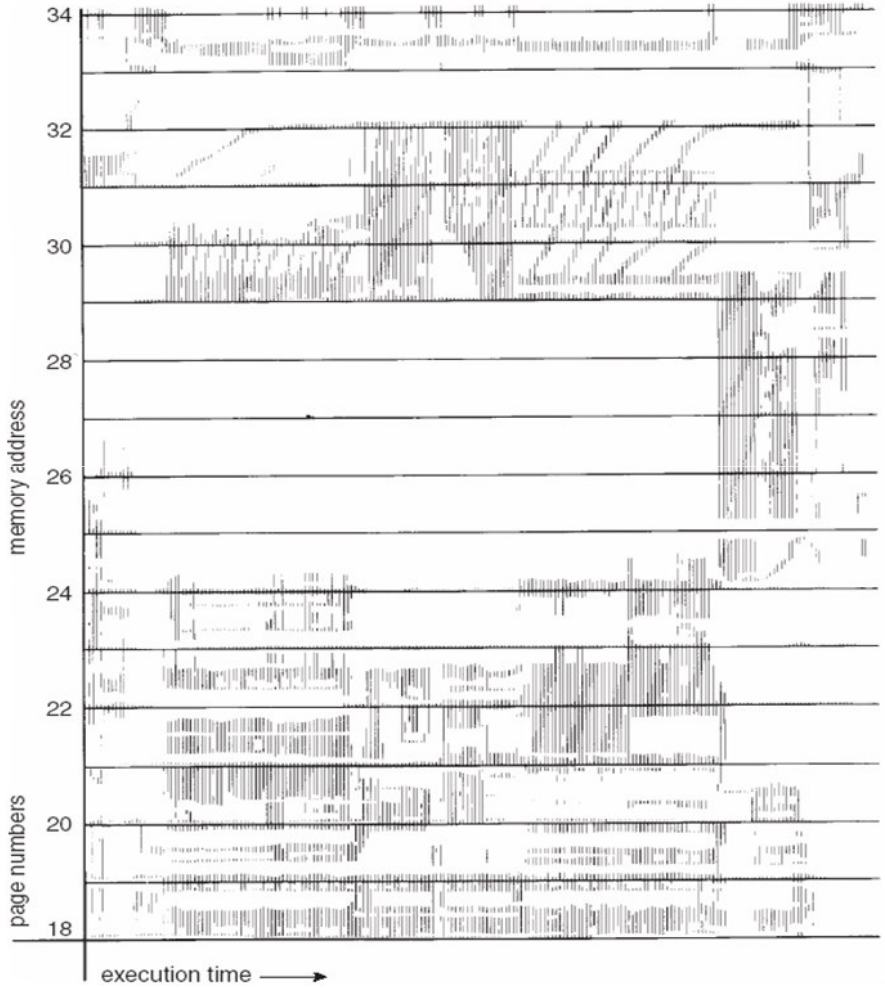
Σ size of locality > total memory size

Page-Fault Frequency Scheme and Frame Allocation for Thrashing Avoidance



- Establish "acceptable" per-process page-fault rate.
  - If actual rate too low, process loses frame.
  - If actual rate too high, process gains frame.

Locality In A Memory-Reference Pattern



Working-Set Model for Thrashing Avoidance

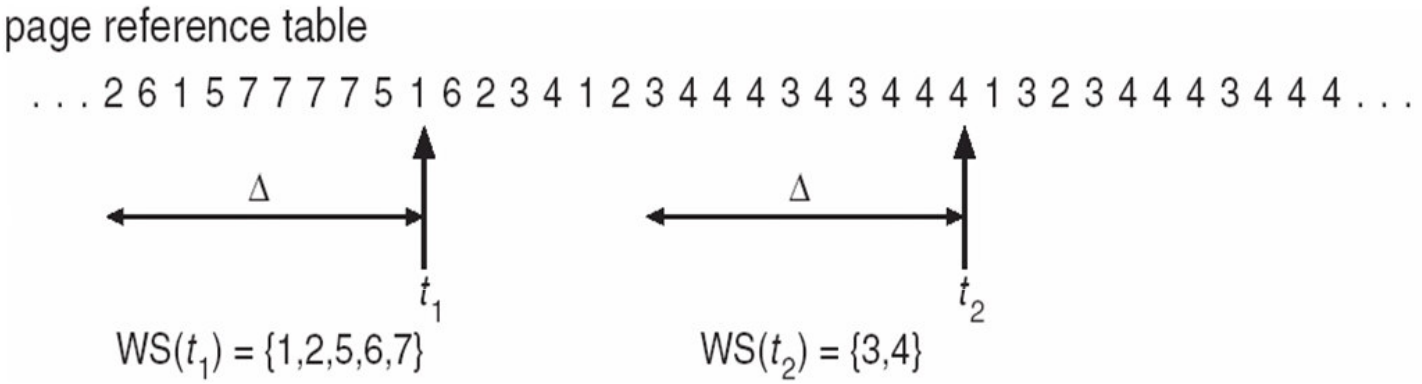
- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  Example: 10,000 instructions
- WSS_i (working set of Process P_i) =
  total number of pages referenced in the most recent Δ references (varies in time)
  - if Δ too small will not encompass entire locality.
  - if Δ too large will encompass several localities.
  - if Δ = unbounded $\Rightarrow$ will encompass entire program.

- $D = \Sigma\ WSS_i \equiv$ total demand for frames

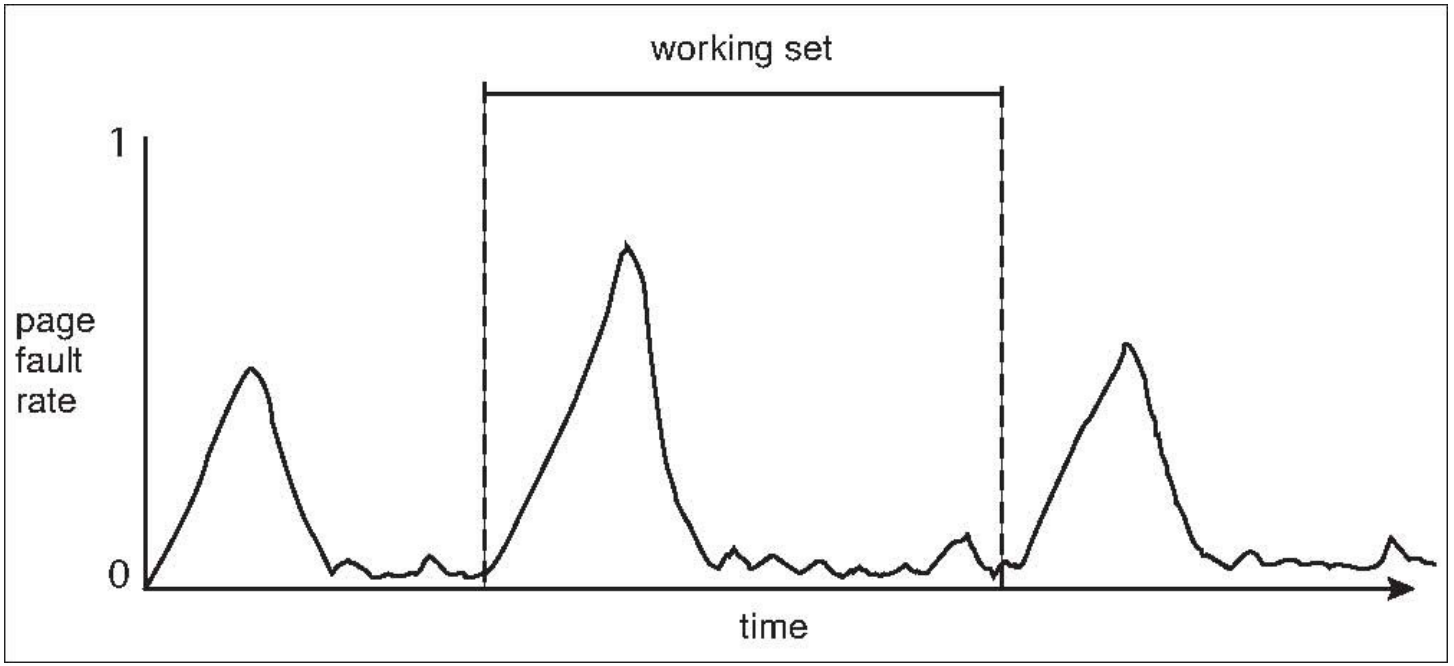
- $D > m \Rightarrow$ Thrashing

- **Policy:** if $D >$ m, then **suspend** some process(es).

Working-set model

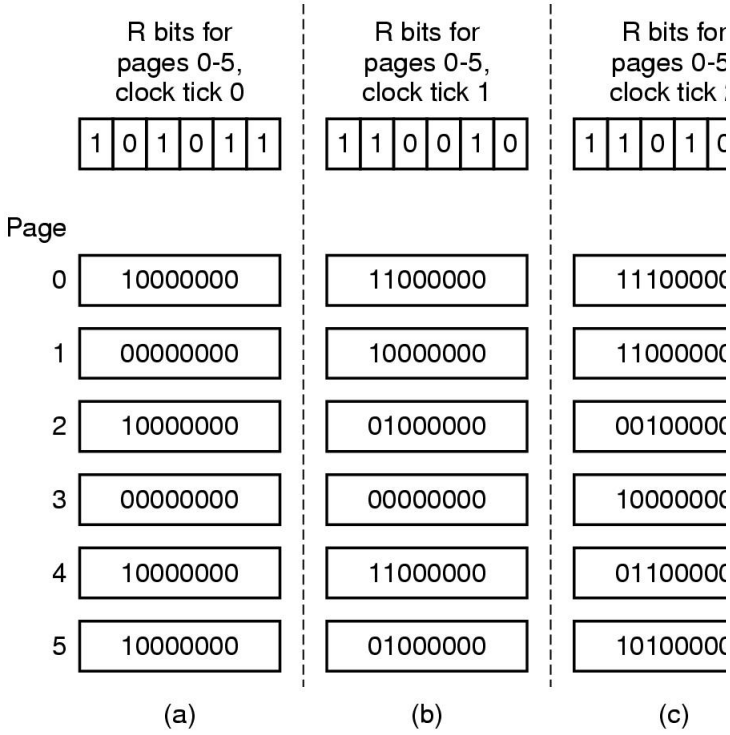


Working Sets and Page Fault Rates



Keeping Track of the Working Set

- Approximate with interval timer + reference bit (recall LRU approximation in software /aging algo)
- Example: Δ = 10,000
  - Timer interrupts after every 5000 time units.
  - Keep in memory 2 bits for each page.
  - Whenever a timer interrupts: copy each page's ref-bit to one of the memory bits and reset each of them
  - If one of the bits in memory = 1 $\Rightarrow$ page in working set.
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units.



Process Suspension for Thrashing Avoidance: which process to chose?

- Lowest priority process
- Faulting process
  - does not have its working set in main memory so will be blocked anyway
- Last process activated
  - this process is least likely to have its working set resident
- Process with smallest resident set
  - this process requires the least future effort to reload
- Largest process
  - obtains the most free frames

Design Issues for Paging Systems

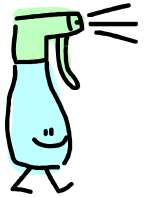
- Global vs local allocation policies
  - Of relevance: Thrashing, working set
- Cleaning Policy
- Fetch Policy
- Page size

Cleaning Policy

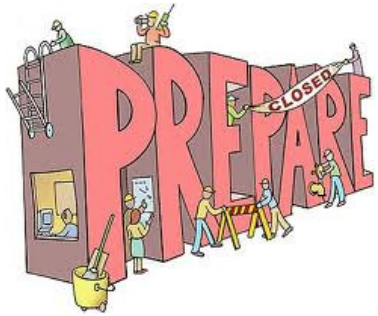
Determines when dirty pages are written to disk:

- Need for a background process, **paging daemon**: periodically inspects state of memory

Precleaning: **first clean then select to free (if needed)**



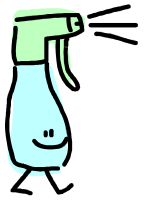
Page buffering: **first free (even when not needed) then clean**



Cleaning Policy: precleaning

Precleaning: first clean then free (if needed)

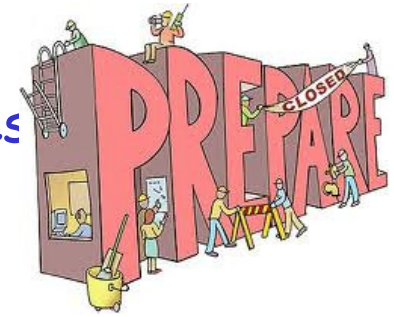
- pages are written out in batches, off-line, periodically: When too few frames are free, paging daemon
  - selects pages to evict using a replacement algorithm
  - can use same circular list (clock)
    - as regular page replacement algorithm but with different pointers

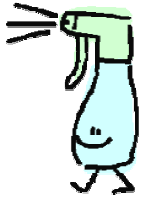


Cleaning Policy: page buffering

Page buffering: **first free then clean**

- use **modified**, **unmodified lists** of replaced pages (freed in advance)
  - A page in the unmodified list may be:
    - reclaimed if referenced again
    - lost when its frame is assigned to another page
  - Pages in the modified list are
    - periodically written out in batches
    - can also be reclaimed





Fetch Policy

Determines when a page should be brought into memory:

Demand paging only brings pages into main memory when a reference is made to it

- Many page faults when process first started

Prepaging brings in more pages than needed

- More efficient to bring in pages that reside contiguously on the disk

Design Issues for Paging Systems

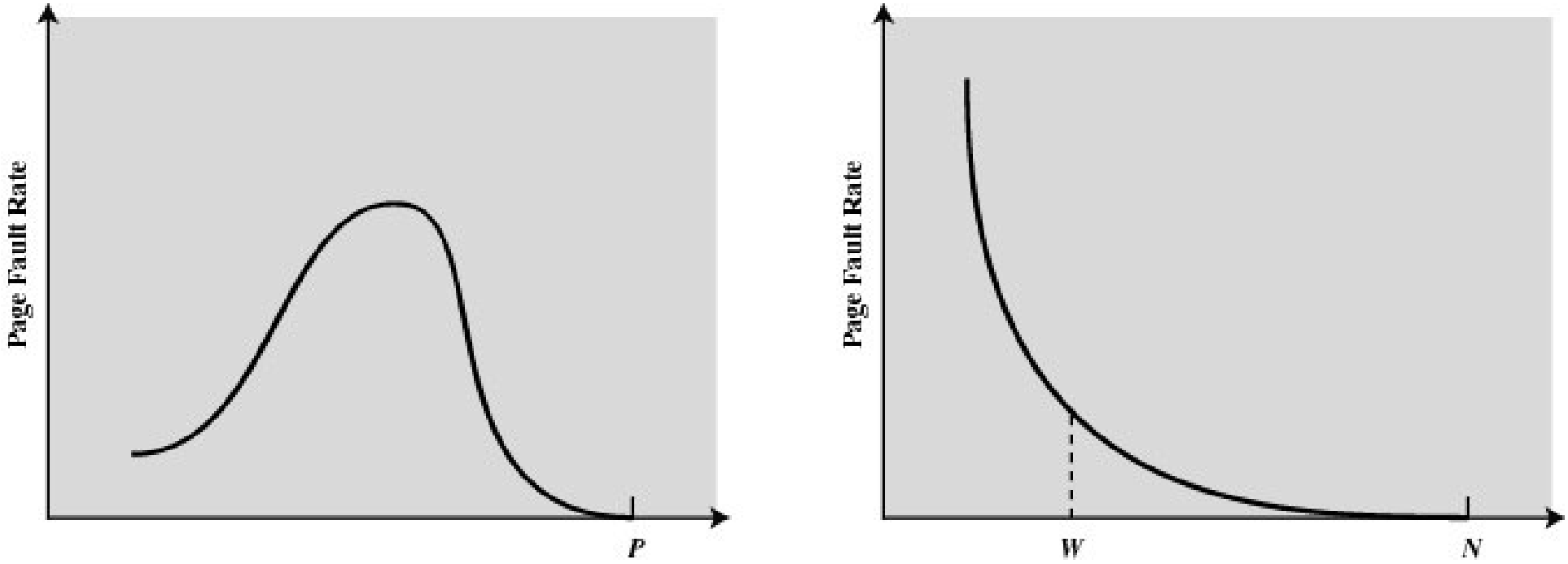
- Global vs local allocation policies
  - Of relevance: Thrashing, working set
- Cleaning Policy
- Fetch Policy
- Page size

Page Size: Trade-off

- Small page size:
  - less internal fragmentation
  - more pages required per process
  - larger page tables (may not be always in main memory)

- Small page size:
  - large number of pages in main memory; as time goes on during execution, the pages in memory will contain portions of the process near recent references. Page faults low.
  - Increased page size causes pages to contain locations further from recent reference. Page faults rise.
  - Page size approaching the size of the program: Page faults low again.

- Secondary memory designed to efficiently transfer large blocks => favours large page size

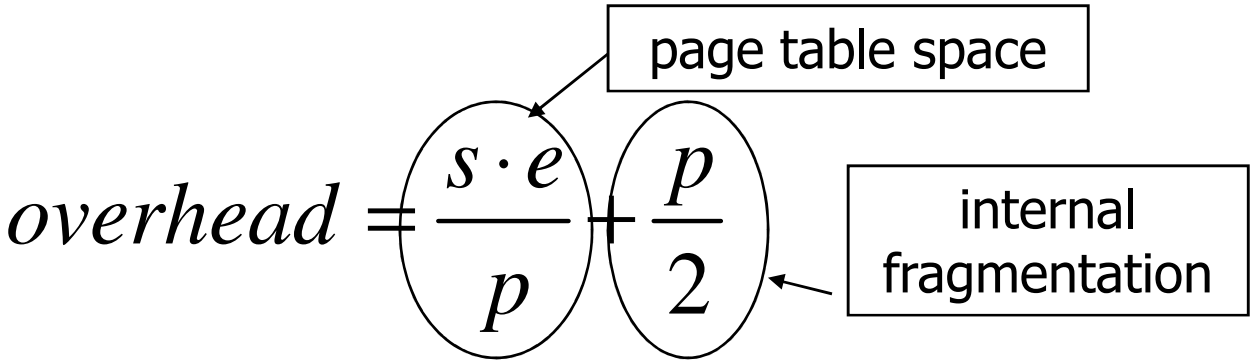


P = size of entire process
W = working set size
N = total number of pages in process

Figure 8.11 Typical Paging Behavior of a Program

Page Size: managing space-overhead trade-off (A.T. MOS2/e)

- Overhead due to page table and internal fragmentation



- Where
  - s = average process size
  - p = page size
  - e = page entry size

Optimized when

$$p = \sqrt{2se}$$

Page Size (cont)

- Multiple page sizes provide the flexibility needed (to also use TLBs efficiently):
  - Large pages can be used for program instructions
  - Small pages can be used for threads

- Multiple page-sizes available by microprocessors: MIPS R4000, UltraSparc, Alpha, Pentium.
- Most current operating systems support only one page size, though.

Implementation Issues

Operating System Involvement with Paging (A.T. MOS2/e)

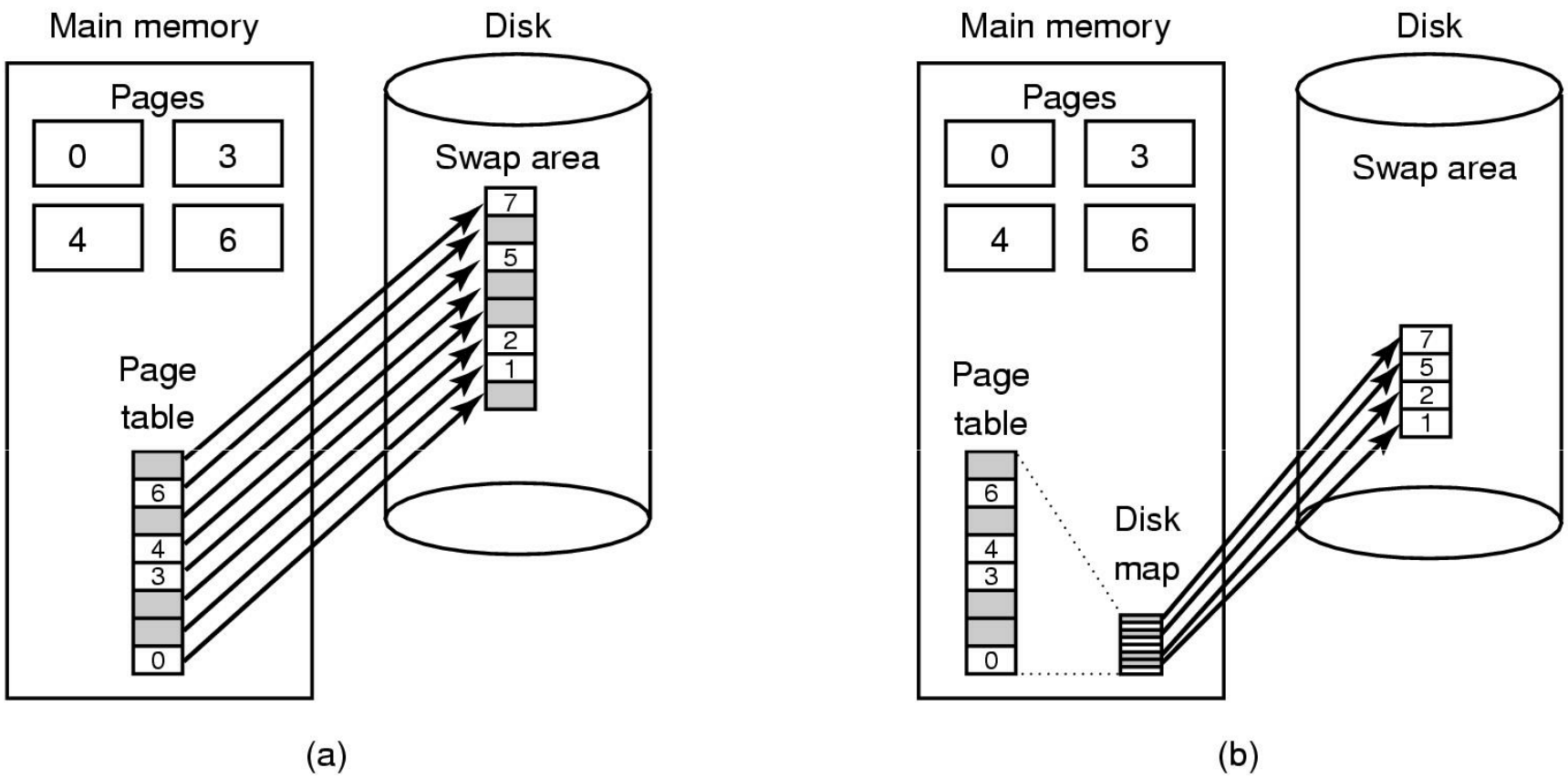
Four times when OS involved with paging

1. Process creation
   - determine program size
   - create page table
2. Process execution
   - MMU reset for new process
   - TLB flushed
3. Page fault time
   - determine virtual address causing fault
   - swap target page out, needed page in
4. Process termination time
   - release page table, pages

Implementation Issues
Locking Pages in Memory

- Need to specify some pages locked, aka pinned (memory interlock)
  - exempted from being target pages
  - Recall lock-bit
- Examples:
  - Proc. has just swapped in a page
    - Or Proc issues call for read from device into buffer
  - another processes starts up
    - has a page fault
  - buffer for the first proc may be chosen to be paged out

Implementation Issues
Backing Store (A.T. MOS2/e)



(a) Paging to static swap area
(b) Backing up pages dynamically

Performance of Demand Paging:

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

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

Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- EAT = (1 – p) x 200 + p (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!!

Other Considerations programmer's perspective

- Program structure
  - Array A[1024, 1024] of integer
  - Each row is stored in one page
  - Program 1

```
for j := 1 to 1024 do
    for i := 1 to 1024 do
        A[i,j] := 0;
```

    1024 x 1024 page faults

  - Program 2

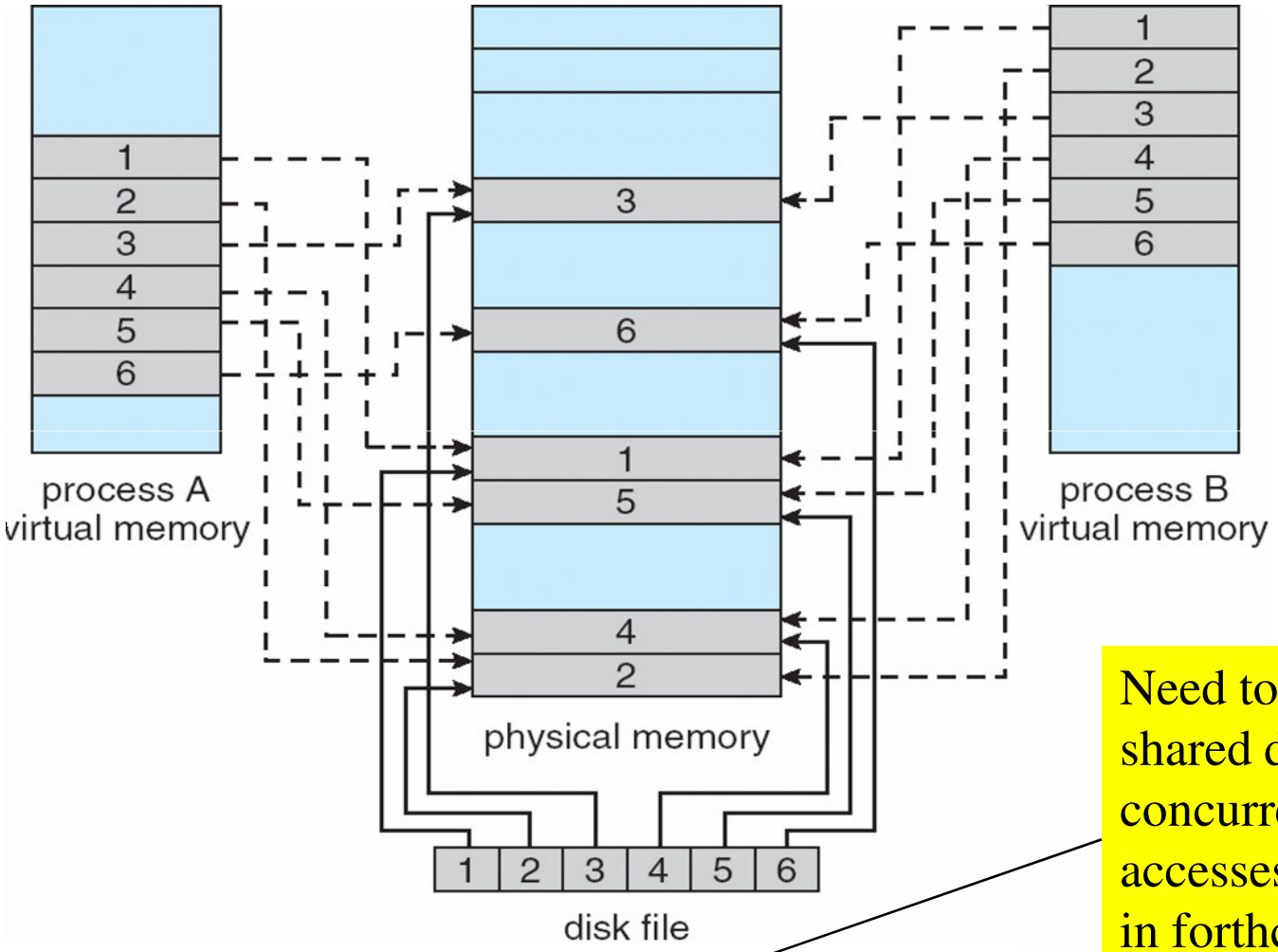
```
for i := 1 to 1024 do
    for j := 1 to 1024 do
        A[i,j] := 0;
```

    1024 page faults

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

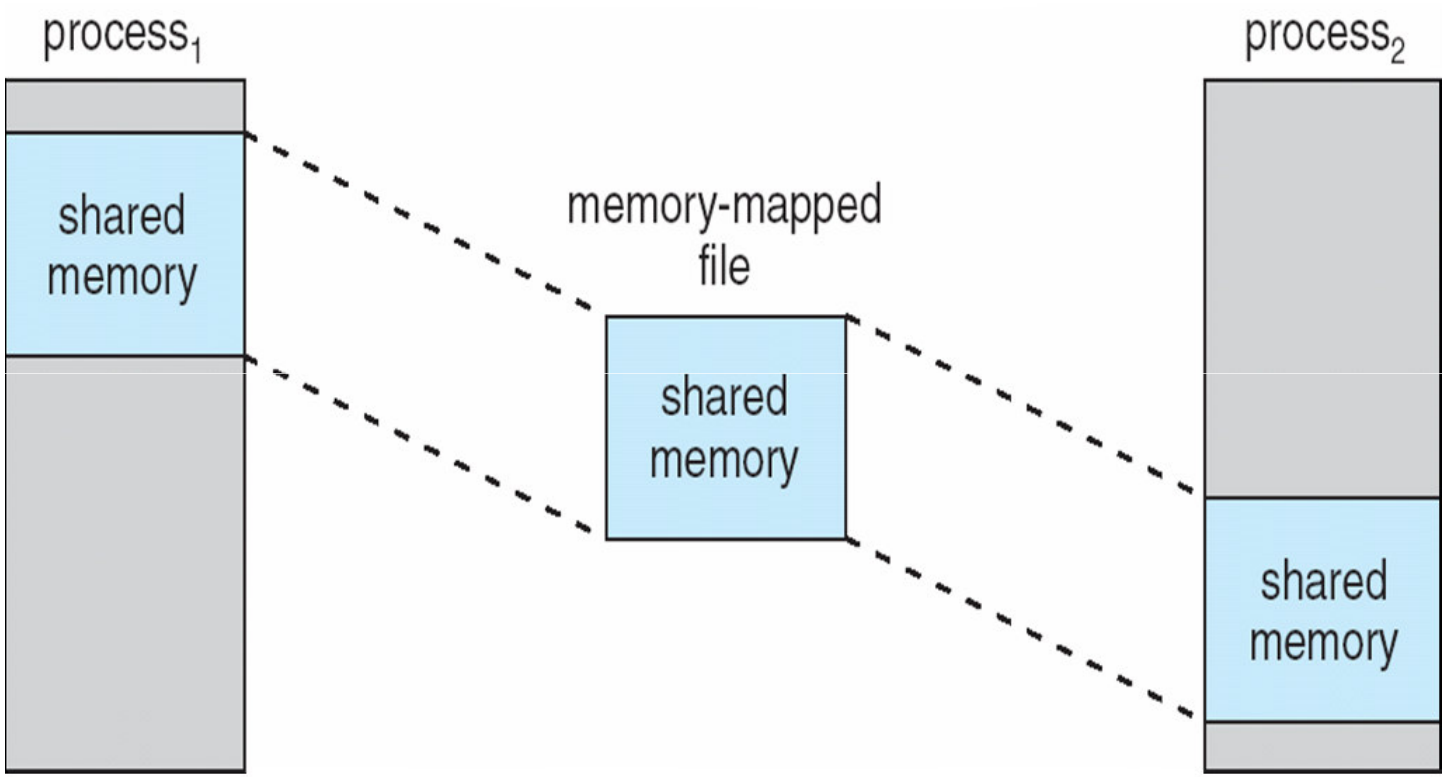
Memory Mapped Files



Can also be used to share data

Need to protect shared data from concurrent accesses; more in forthcoming lectures

Memory-Mapped Shared Memory in Windows



THE way to share data among processes in windows systems

Operating System Examples highlight issues

- Windows XP
- Solaris

Windows XP virtual memory

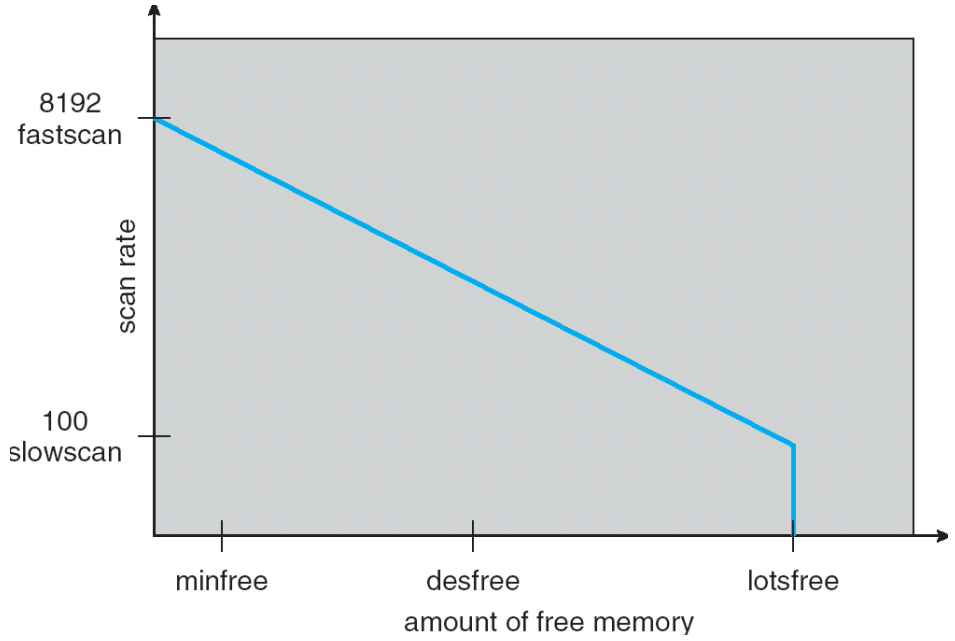
- Uses demand paging with **clustering**. Clustering brings in pages surrounding the faulting page

- Processes are assigned **working set minimum** and **working set maximum**
  - Working set minimum: 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:
    - Allocate from the free-list if non-empty

- 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
  - trimming removes pages from processes that have pages in excess of their working set minimum (victims selected using second-chance or FIFO -like)

Solaris virtual memory

- Maintains a list of free (but not overwritten) pages to assign to faulting processes (prepare then clean)
  - *Lotsfree* – threshold parameter (amount of free memory) to begin paging
  - *Desfree* – threshold parameter to increasing paging
  - *Minfree* – threshold parameter to begin swapping
- Paging is performed by *pageout* process: scans pages using modified clock algorithm (2 hands: second-chance and freeing hands)
  - *Scanrate* : ranges from *slowscan* to *fastscan* - depending upon the amount of free memory available



Solaris 2 Page Scanner