CSE 3320
Operating Systems
Page Replacement Algorithms and Segmentation

Jia Rao

Department of Computer Science and Engineering

http://ranger.uta.edu/~jrao

Recap of last Class

- Virtual memory
  - Memory overload
  - What if the demanded memory page is not in memory?
    - Paging
  - What if some other pages need to be kicked out?
    - Page replacement algorithms
- A single virtual address per process
  - Multiple virtual addresses → segmentation

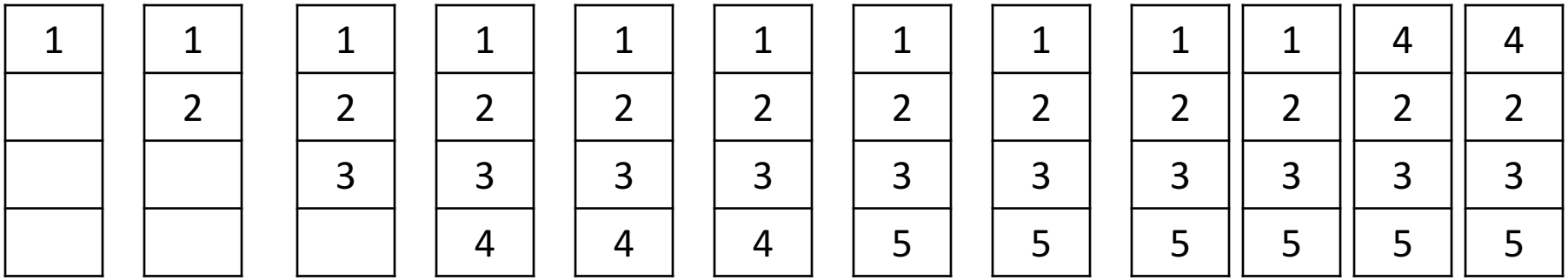
Page Replacement Algorithms

- Like cache miss, a *page fault* forces choice
  - which page must be removed
  - make room for incoming pages
- Modified page must first be saved
  - Modified/Dirty bit
  - unmodified just overwritten
- Better not to choose an often used page
  - will probably need to be brought back in soon
  - Temporal locality
- Metrics
  - Low page-fault rate

Optimal Page Replacement Algorithm

- Replace page needed at the farthest point in future
  - Optimal but unrealizable
  - OS has to know when each of the pages will be referenced next
  - Good as a benchmark for comparison
    - Take two runs, the first run gets the trace, and the second run uses the trace for the replacement
    - Still, it is *only* optimal with respect to that specific program

Reference string: 1 2 3 4 1 2 5 1 2 3 4 5 6 page faults



Not Recently Used (NRU)

- Each page has R bit (referenced) and M bit (modified)
  - bits are *set* when page is referenced and modified
  - OS clears R bits periodically (by clock interrupts)
- Pages are classified
  1. not referenced, not modified
  2. not referenced, modified
  3. referenced, not modified
  4. referenced, modified
- NRU removes a page at random
  - From the lowest numbered non-empty class

FIFO Page Replacement Algorithm

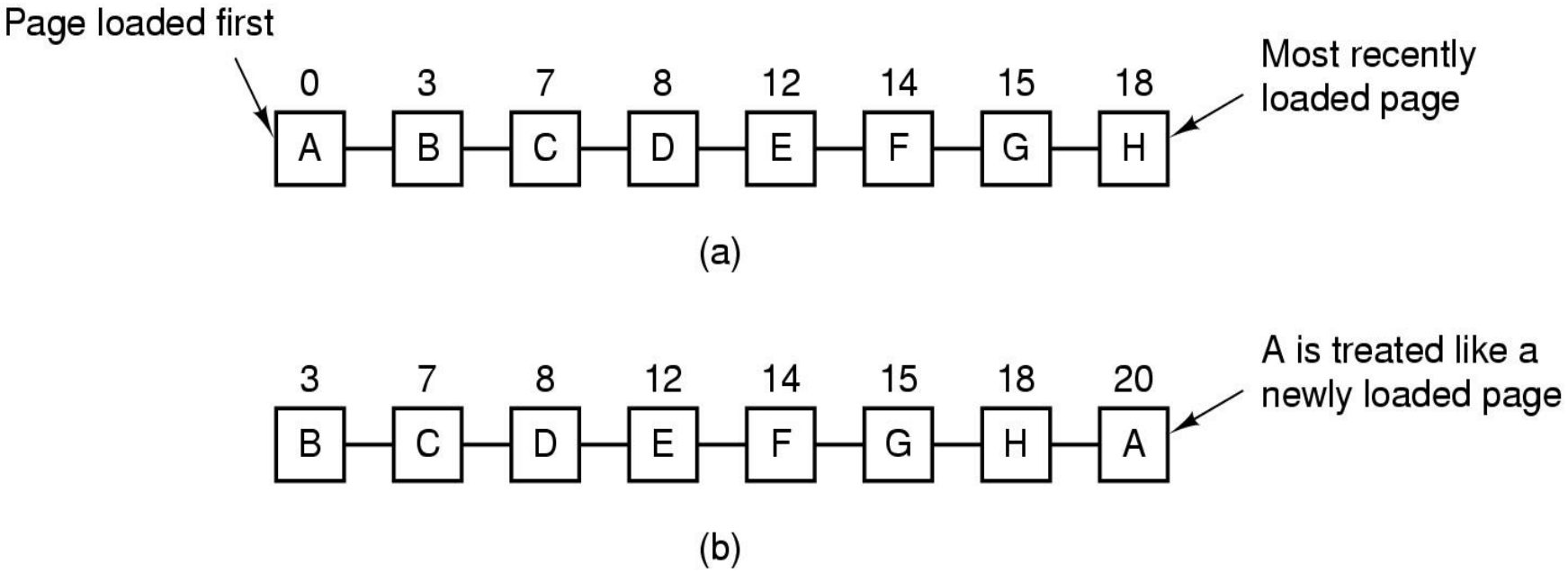
- Maintain a linked list of all pages
  - in the order they came into memory
- Page at beginning of list replaced (the oldest one)
- Disadvantage
  - page in memory the longest (oldest) may be often used

Reference string: 1 2 3 4 1 2 5 1 2 3 4 5

10 page faults

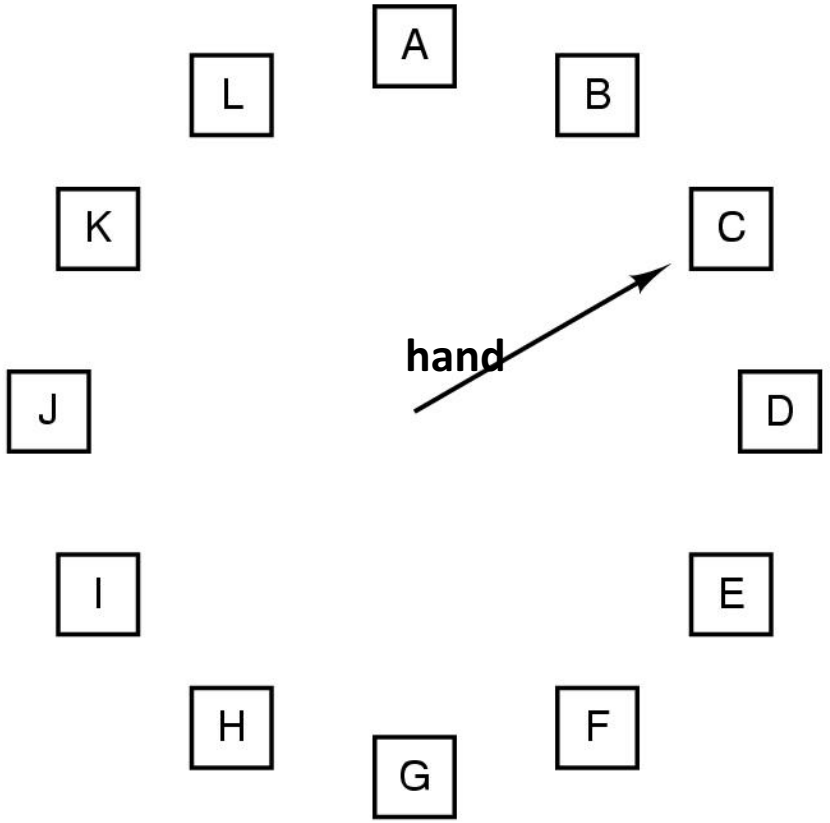
1	1	1	1	1	1	5	5	5	5	4	4
	2	2	2	2	2	2	1	1	1	1	5
		3	3	3	3	3	3	2	2	2	2
			4	4	4	4	4	4	3	3	3

Second Chance Page Replacement Algorithm



- OS clears R bits periodically (by clock interrupts)
- Second chance (FIFO-extended): looks for an *oldest* and *not referenced* page in the previous clock interval; if all referenced, FIFO
  - (a) pages sorted in FIFO order
  - (b) Page list if a page fault occurs at time 20, and *A* has *R* bit set (numbers above pages are loading times);
  - (c) what if *A* has R bit cleared?

The Clock Page Replacement Algorithm



When a page fault occurs, the page the hand is pointing to is inspected. The action taken depends on the R bit:

R = 0: Evict the page

R = 1: Clear R and advance hand

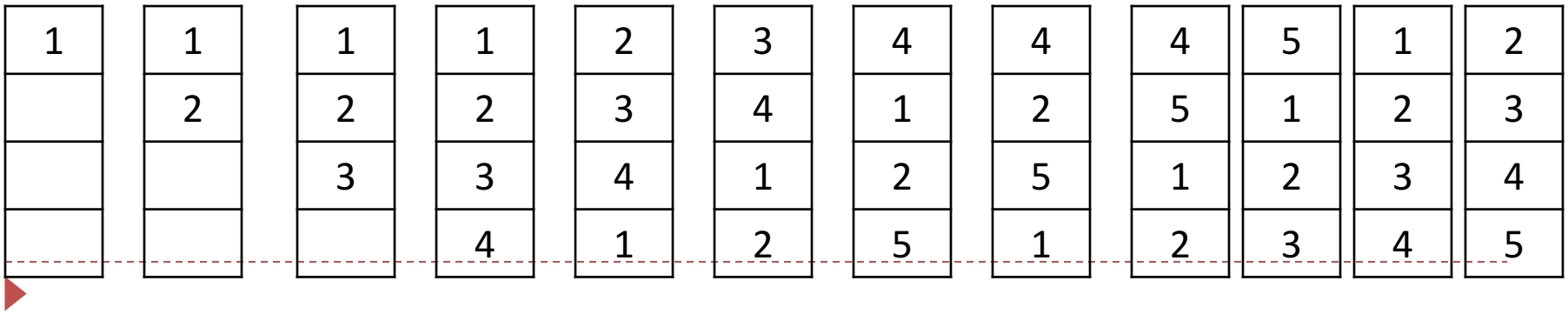
- The clock page replacement algorithm differs Second Chance only in the implementation
  - No need to move pages around on a list
  - Instead, organize a circular list as a clock, with a hand points to the oldest page

Least Recently Used (LRU)

- Assume pages used recently will be used again soon
  - throw out page that has been least used recently
- Must keep a linked list of pages
  - most recently used at front, least at rear
  - update this list every memory reference !!!
    - finding, removing, and moving it to the front
- Special hardware:
  - Equipped with a 64-bit counter
  - keep a counter field in each page table entry
  - choose page with lowest value counter
  - periodically zero the counter (NRU)
  - And more simulation alternatives

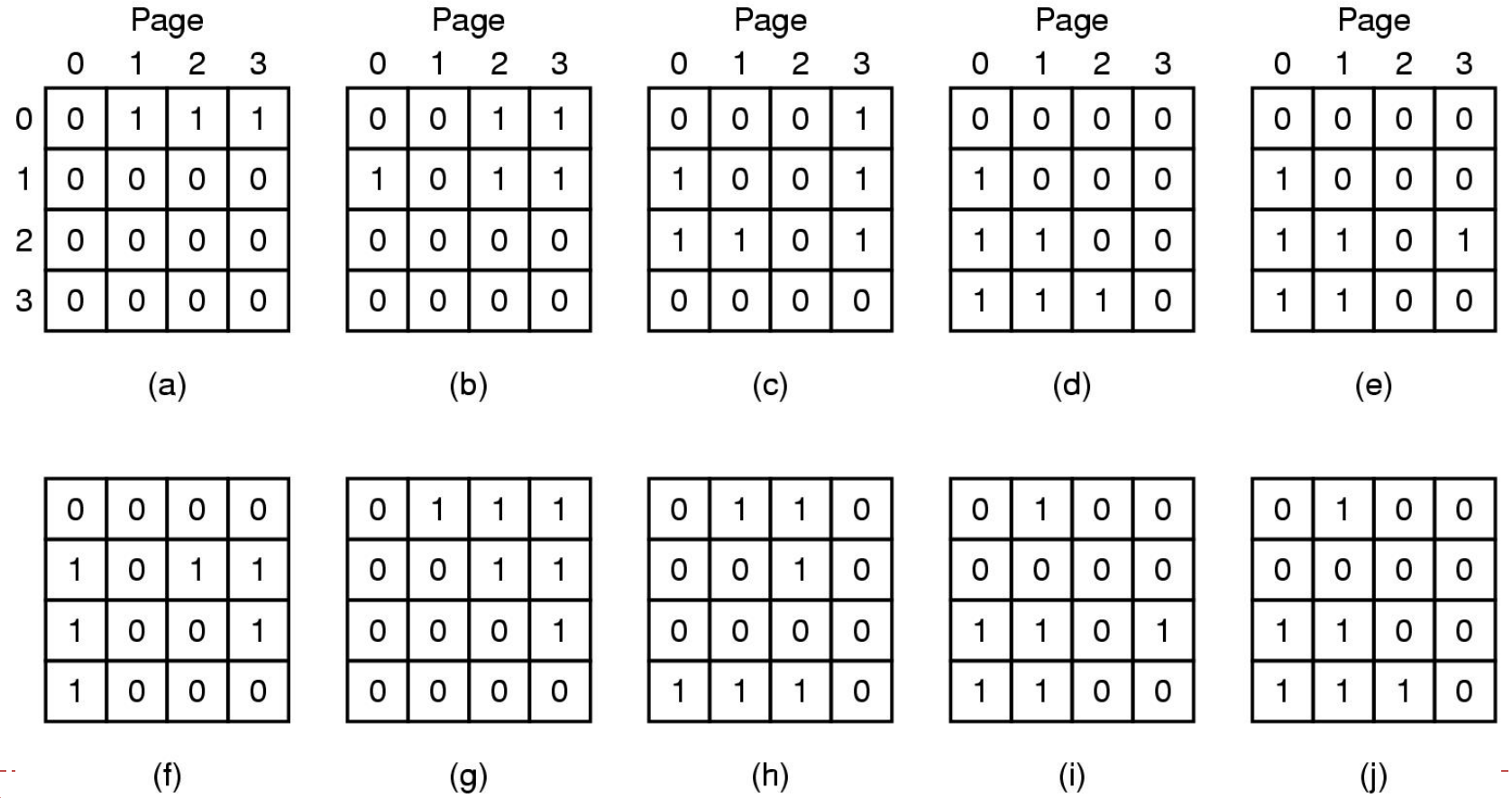
Reference string: 1 2 3 4 1 2 5 1 2 3 4 5

8 page faults



LRU Support in Hardware

- For a RAM with *n* page frames, maintain a matrix of *n x n* bits; set all bits of row k to 1, and then all bits of column k to 0. At any instant, the row whose binary value is lowest is the least recently used. Page reference order: 0 1 2 3 2 1 0 3 2 3

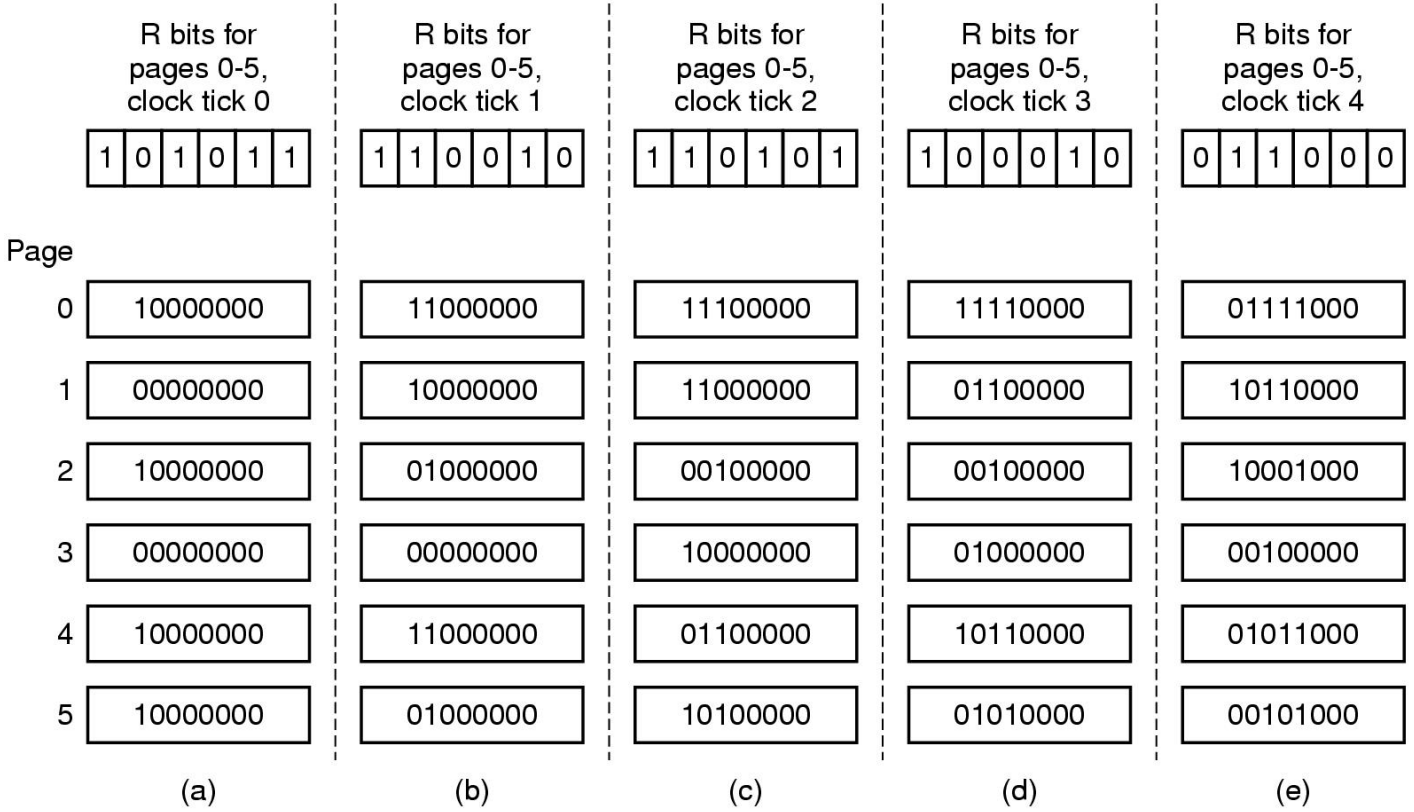


Not Frequently Used (NFU)

- NFU (Not Frequently Used): uses a software counter per page to track how *often* each page has been referenced, and chose the least to kick out
  - OS adds R bit (0 0r 1) to the counter at each clock interrupt
  - Problem: never forgets anything

Aging - Simulating LRU/NFU in Software

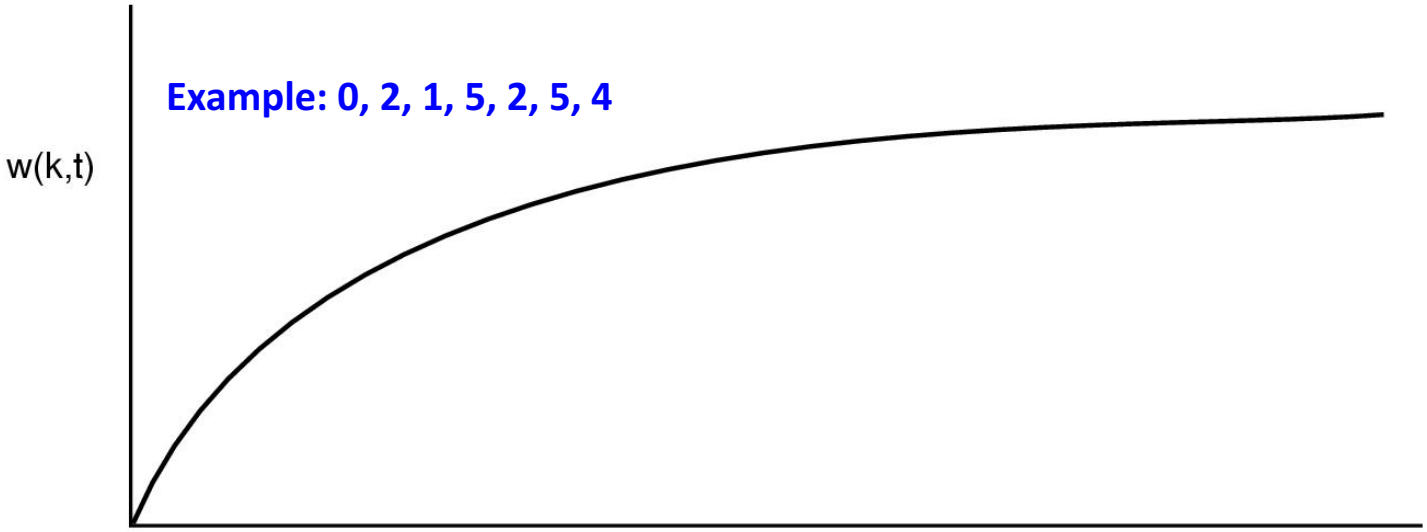
- **Aging: the counters are each shifted right 1 bit before the R bit is added in; the R bit is then added to the leftmost**
  - The page whose counter is the lowest is removed when a page fault



The *aging* algorithm simulates LRU in software, 6 pages for 5 clock ticks, (a) – (e)

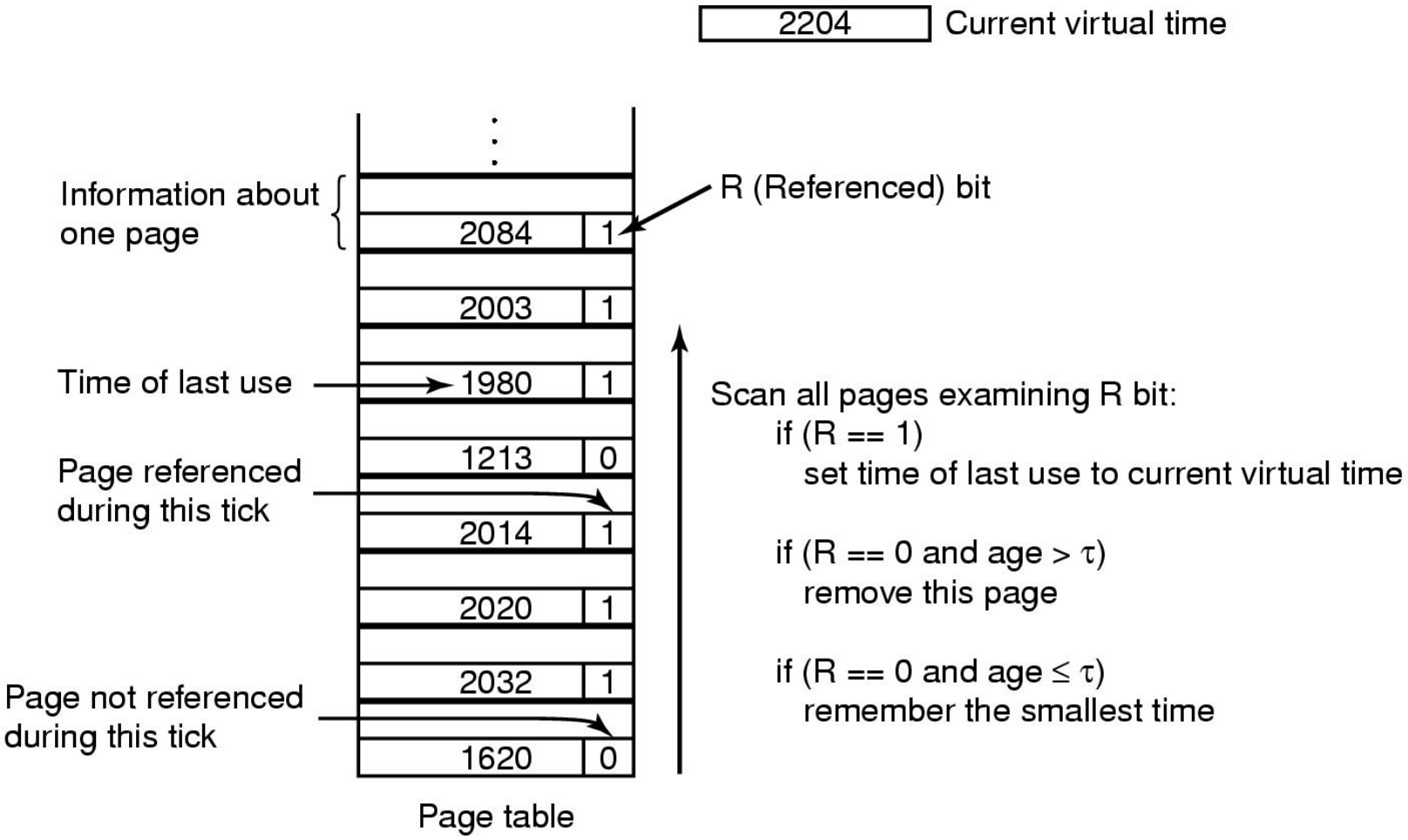
The Working Set and Pre-Paging

- Demand paging vs. pre-paging
- Working set: the set of pages that a process is *currently* using
- Thrashing: a program causing page faults every a few instructions
- Observation: working set does not change quickly due to locality
  - Pre-paging working set for processes in multiprogramming



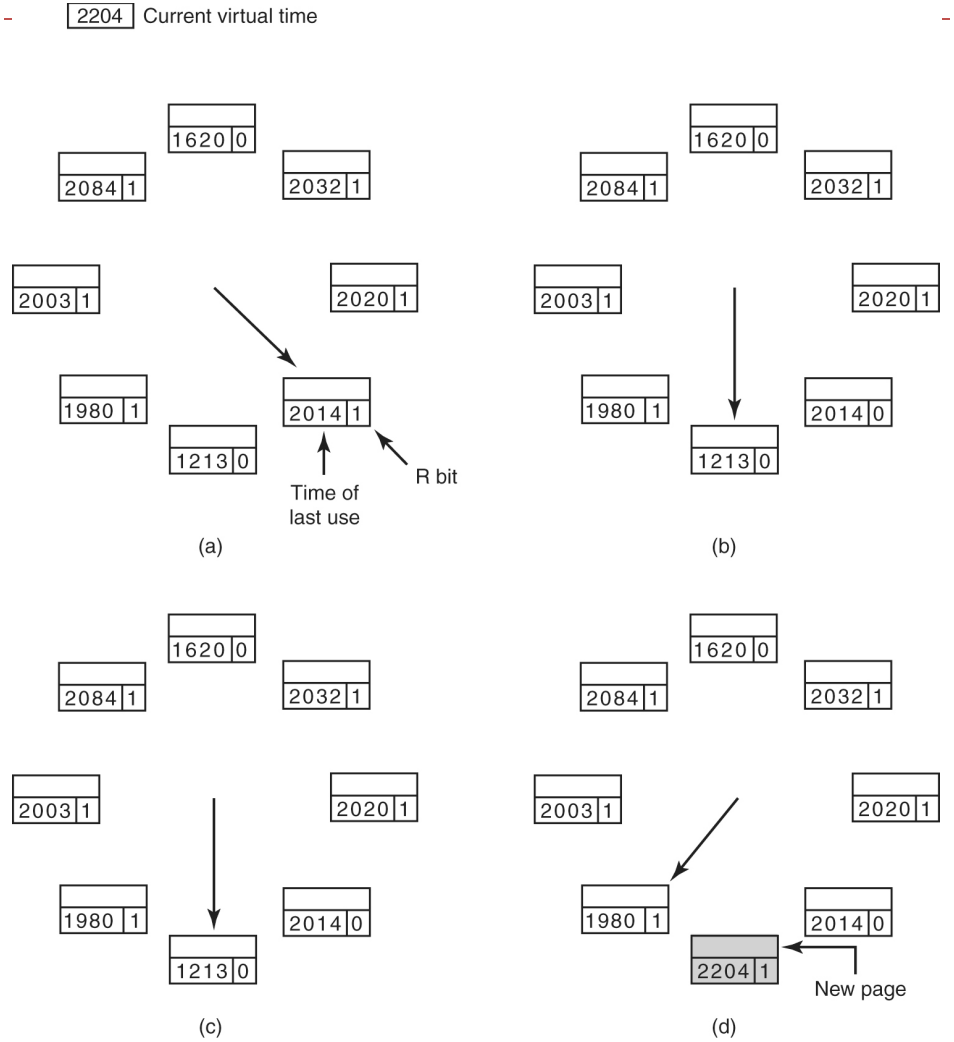
The *working set* is the set of pages used by the *k* most recent memory references
w(k,t) is the size of the working set at time, *t*

The Working Set Page Replacement Algorithm



The working set algorithm

The WSClock Page Replacement Algorithm



Operation of the WSClock algorithm

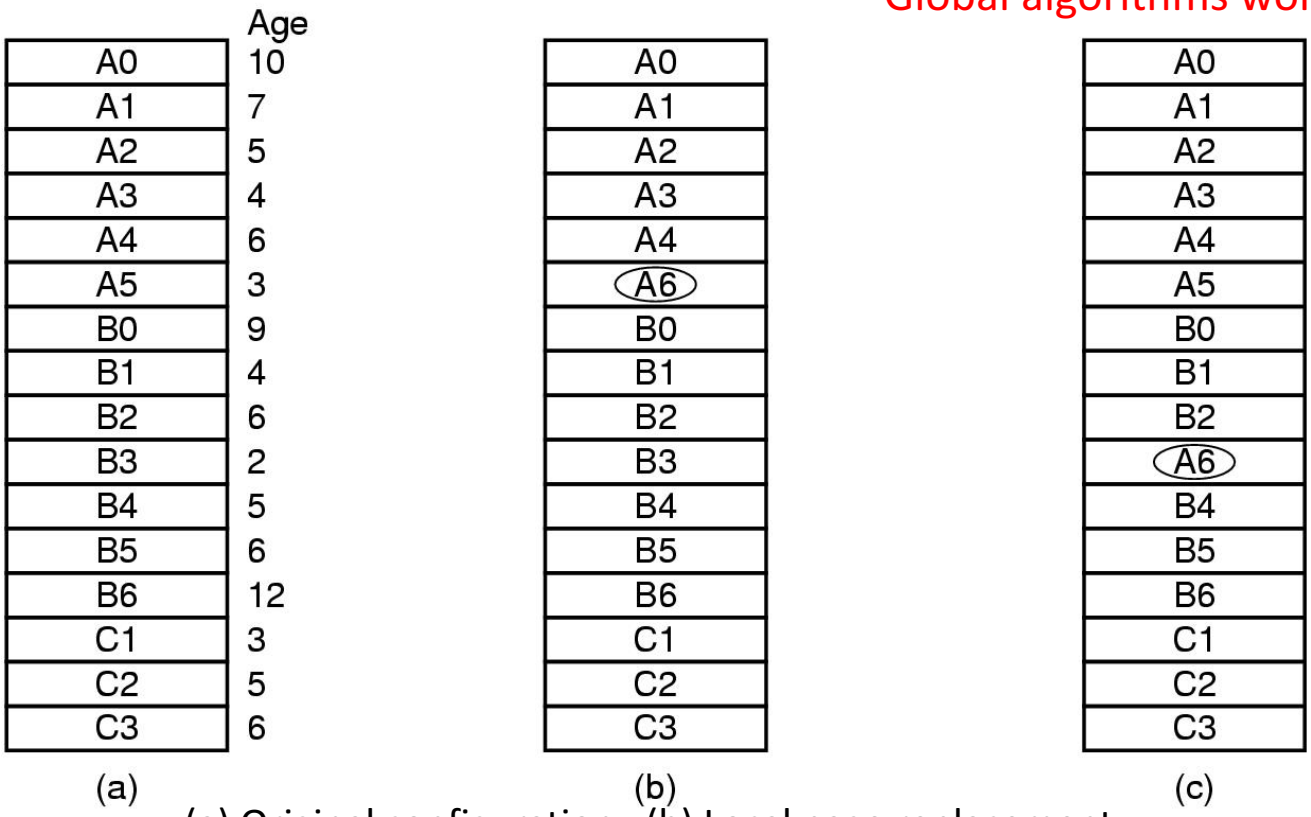
Review of Page Replacement Algorithms

Algorithm	Comment
Optimal	Not implementable, but useful as a benchmark
NRU (Not Recently Used)	Very crude
FIFO (First-In, First-Out)	Might throw out important pages
Second chance	Big improvement over FIFO
Clock	Realistic
LRU (Least Recently Used)	Excellent, but difficult to implement exactly
NFU (Not Frequently Used)	Fairly crude approximation to LRU
Aging	Efficient algorithm that approximates LRU well
Working set	Somewhat expensive to implement
WSClock	Good efficient algorithm

Design Issues for Paging Systems

- Local page replacement vs. global page replacement
  - How is memory allocated among the competing processes?

Global algorithms work better



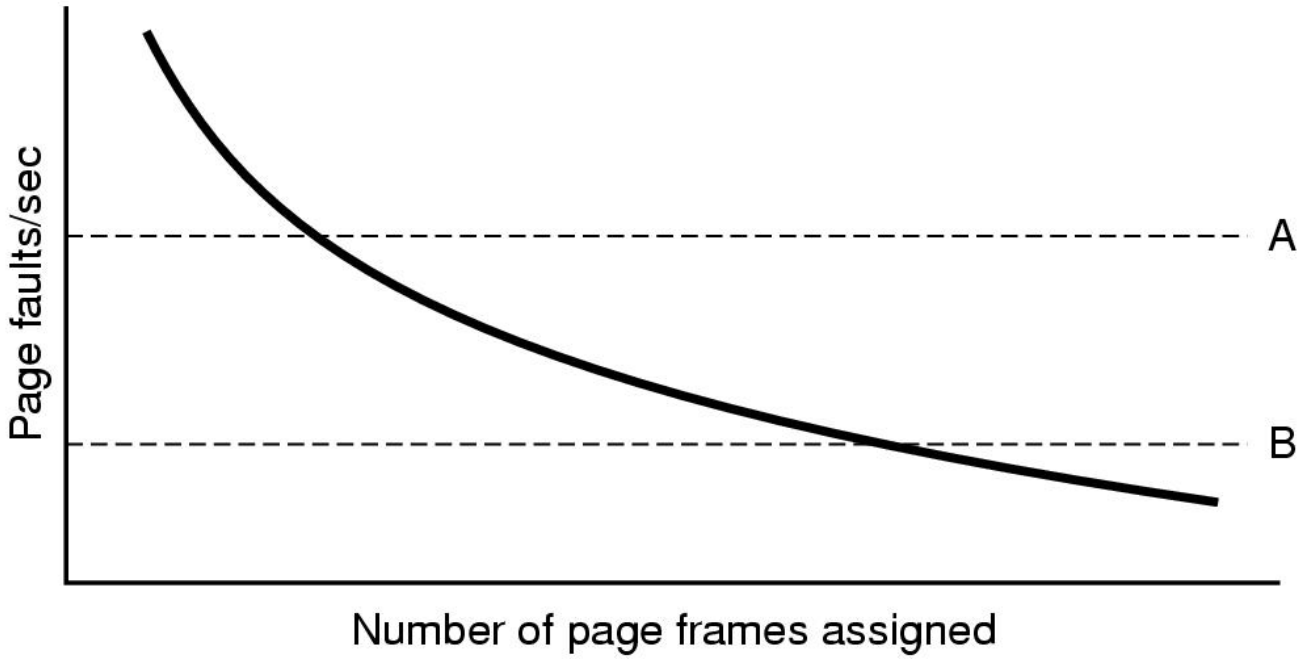
(a) Original configuration. (b) Local page replacement.
(c) Global page replacement.

Design Issues for Paging Systems (2)

- Local page replacement: static allocation
  - What if the working set of some process grows?
  - What if the working set of some process shrinks?
  - The consideration: thrashing and memory utilization
- Global page replacement: dynamic allocation
  - How many page frames assigned to each process
    - Keep monitoring the working set size
    - Allocating an equal share of available page frames
    - Allocating a proportional share of available page frames
    - Or hybrid allocation, using PFF (page fault frequency)

Page Fault Frequency (PFF)

- PFF: control the size of allocation set of a process
  - when and how much to increase or decrease a process' page frame allocation
- Replacement: what page frames to be replaced



Page fault rate as a function of the number of page frames assigned

Load Control

- Despite good designs, system may still have thrashing
  - When combined *working sets* of all processes exceed the capacity of memory
- When PFF algorithm indicates
  - some processes need more memory
  - but no processes need less
- Solution :
  - swap one or more to disk, divide up pages they held
  - reconsider degree of multiprogramming
    - CPU-bound and I/O-bound mixing

Reduce number of processes competing for memory

Page Size (1)

Small page size

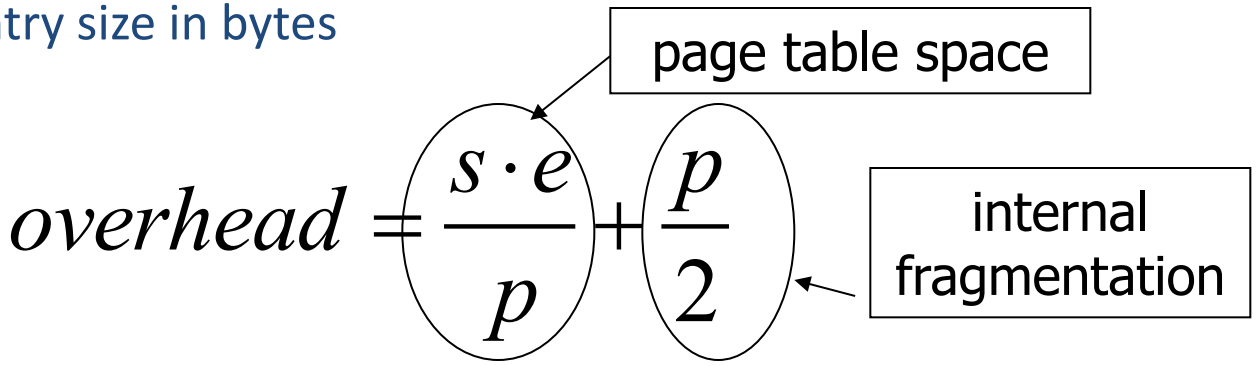
- Advantages
  - less unused program in memory (due to *internal fragmentation*)
  - better fit for various data structures, code sections
- Disadvantages
  - programs need many pages, larger page tables
  - Long access time of page (compared to transfer time)
  - Also maybe more paging actions due to page faults

Page Size (2)

- Tradeoff: overhead due to page table and internal fragmentation

  Where

  - s = average process size in bytes
  - p = page size in bytes
  - e = page entry size in bytes



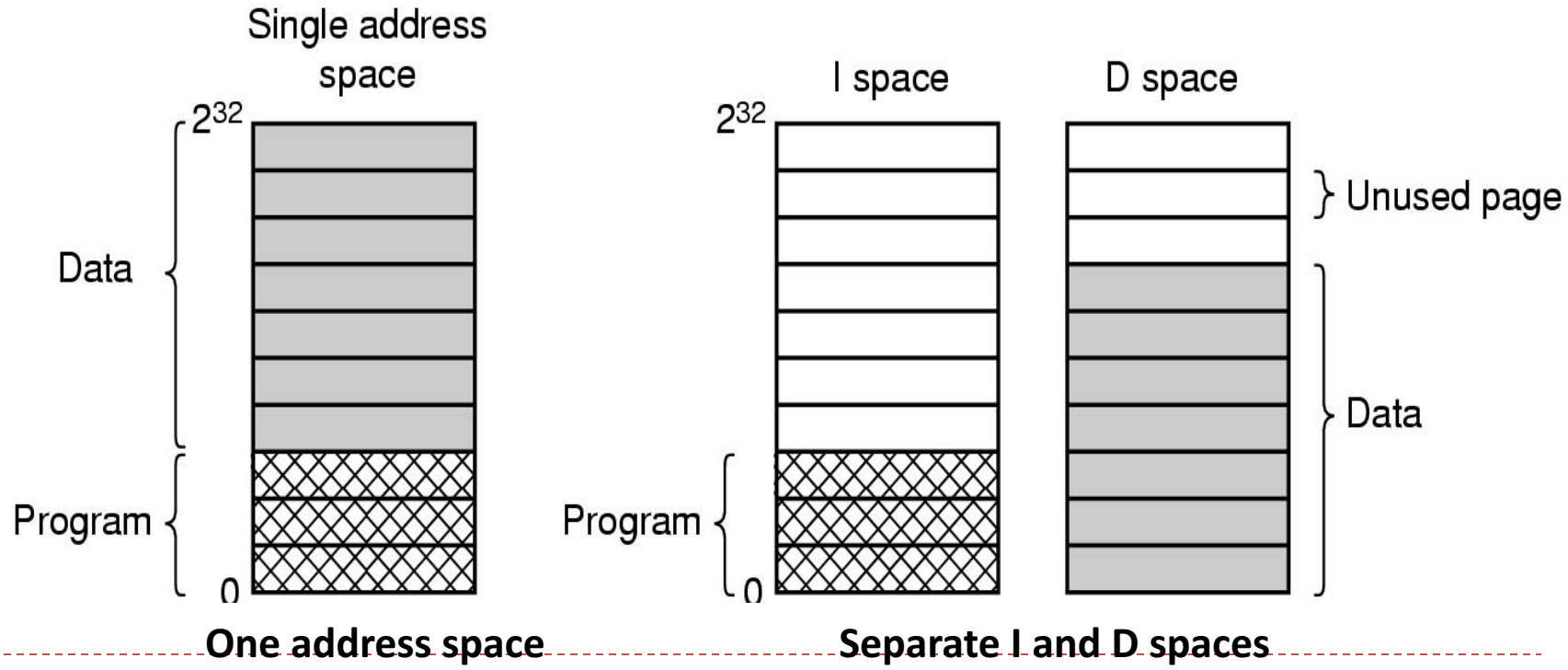
Optimized/minimized when f' (p)=0

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

s=1M, e=8B → p=4KB

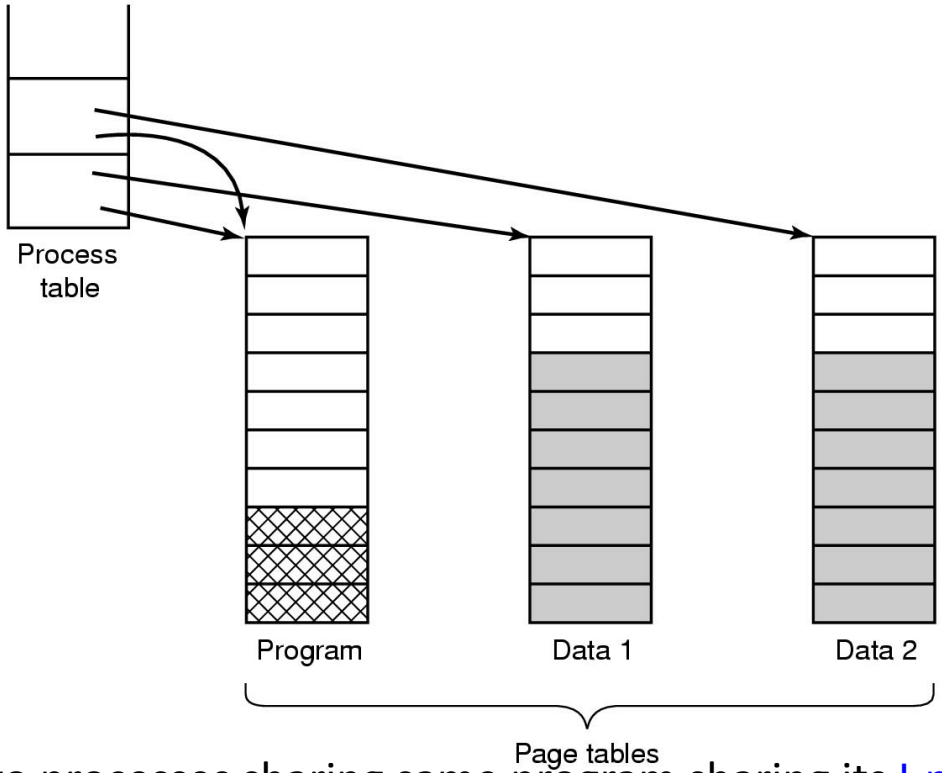
Separate Instruction and Data Spaces

- What if the single virtual address space is not enough for *both* program and data?
  - Doubles the available virtual address space, and ease page sharing of multiple processes
  - Both addr. spaces can be paged, each has its own page table



Shared Pages

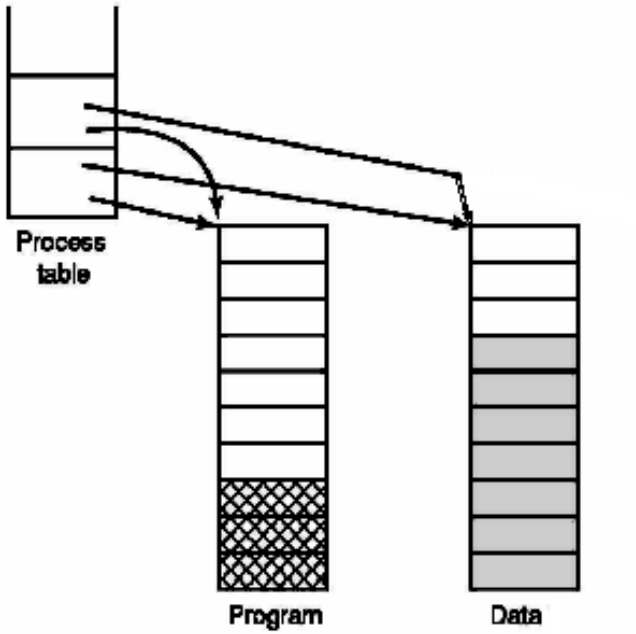
- How to allow multiple processes to share the pages when running the same program at the same time?
  - One process has its own page table(s)



Two processes sharing same program sharing its I-page table

Shared Pages (2)

- What to do when a page replacement occurs to a process while other processes are sharing pages with it? Minor page faults.
- How sharing data pages, is compared to sharing code pages?
- UNIX fork() and *copy-on-write*
  - *Generating a new page table point to the same set of pages, but not duplicating pages until…*
  - *A violation of read-only causes a trap*



Cleaning Policy

- Need for a background process, *paging daemon*
  - periodically inspects state of memory
  - To ensure plenty of free page frames
- When too few frames are free
  - selects pages to evict using a replacement algorithm
- Write back policy
  - Write dirty pages back when the ratio of dirty pages exceeds a threshold

    /proc/sys/vm/dirty_ratio

Implementation Issues

Four times when OS involved with paging

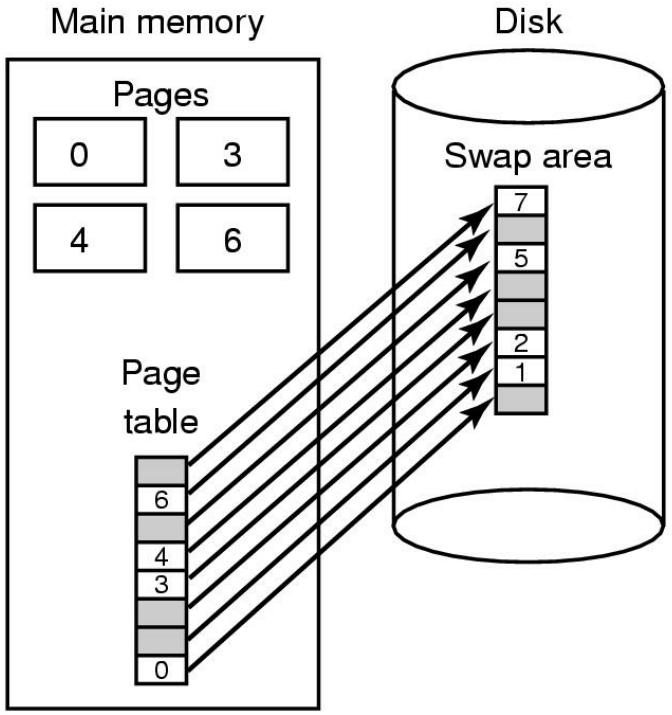
1. Process creation
   - determine program size
   - create page table
2. Process execution
   - MMU reset for new process
   - TLB flushed (as invalidating the cache)
3. Page fault time
   - determine virtual address causing fault
   - swap target page out, needed page in
4. Process termination time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

Page Fault Handling

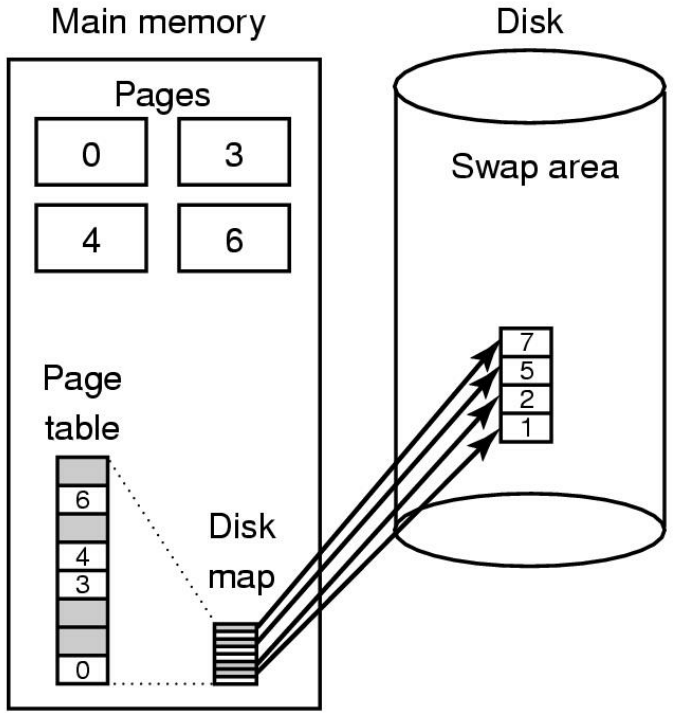
1. Hardware traps to kernel
2. General registers saved
3. OS determines which virtual page needed
4. OS checks validity of address, seeks page frame
5. If selected frame is dirty, write it to disk
6. OS brings the new page in from disk
7. Page tables updated
8. Faulting instruction backed up to when it began
9. Faulting process scheduled
10. Registers restored
11. Program continues

Backing Store – Disk Management

- How to allocate page space on the disk in support of VM?
  - Static swap area (pages copied): adding the offset of the page in the virtual address space to the start of the swap area
  - Dynamic swapping (page not copied, a table-per-process needed)



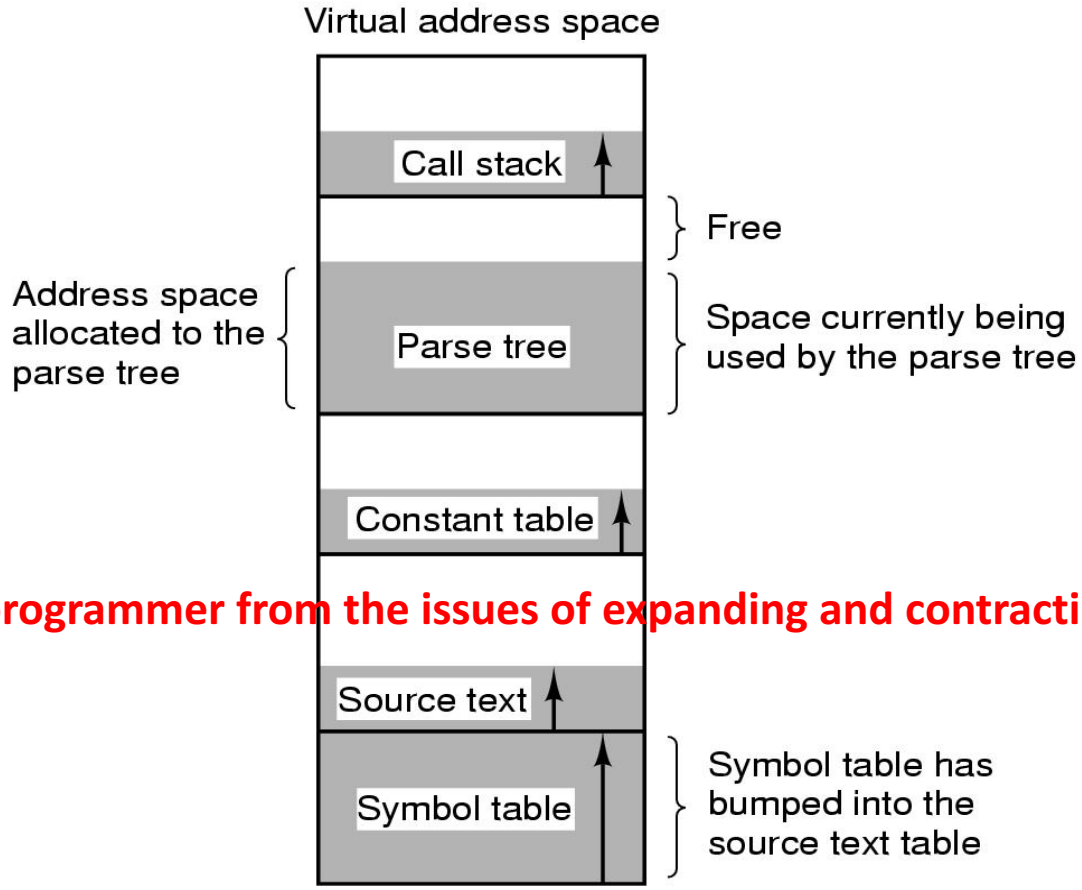
(a) Paging to *static swap area*.



(b) Backing up pages dynamically

Segmentation (1)

- Why to have two or more separate virtual address spaces?

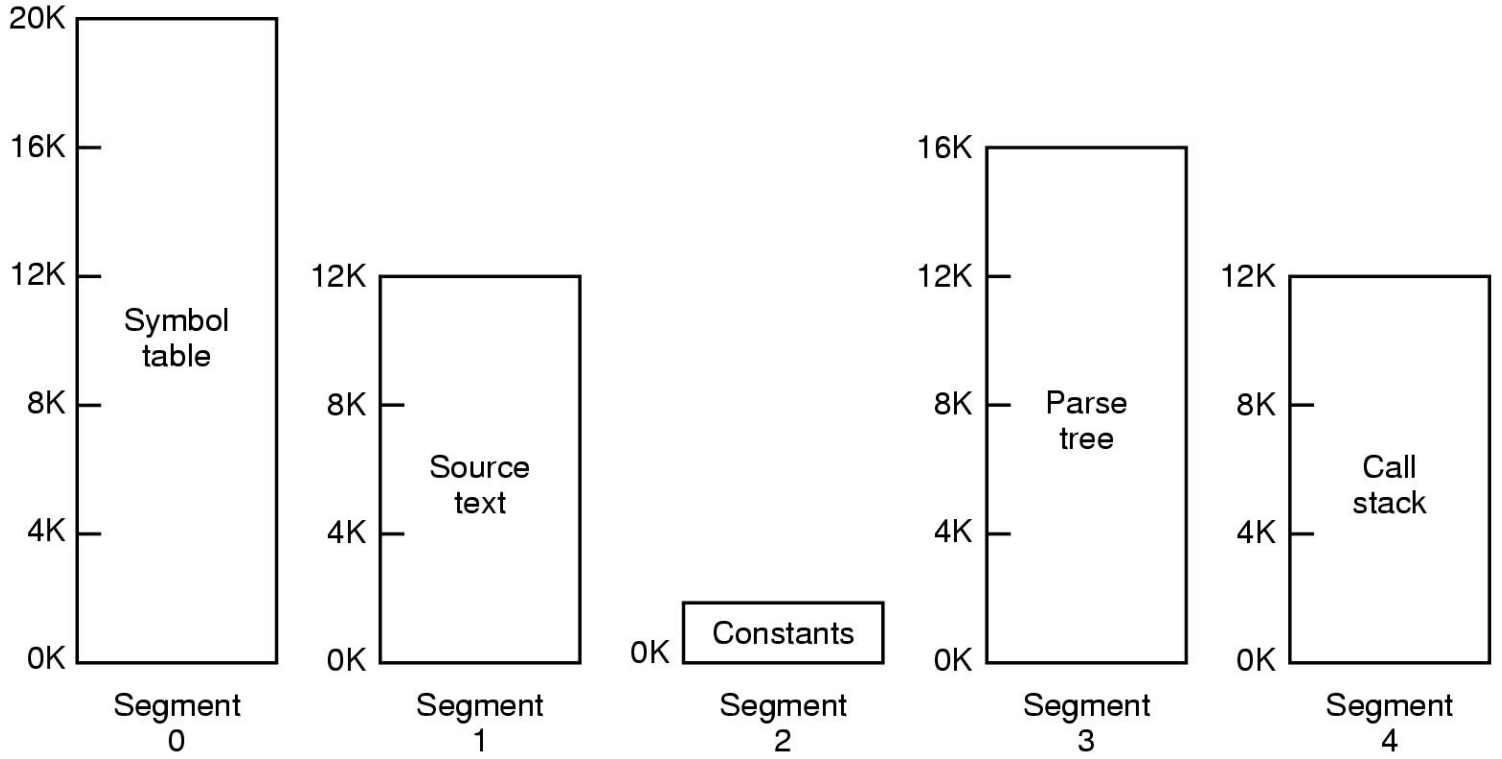


How to free a programmer from the issues of expanding and contracting tables?

- One-dimensional address space with *growing* tables for compiling, one table may bump/interfere into another

Segmentation (2)

- Segments: many *independent* virtual address spaces
  - A logical entity, known and used by the programmer
  - Two-dimensional memory: the program must supply a two-part address, *a segment number* and *an address within an segment*



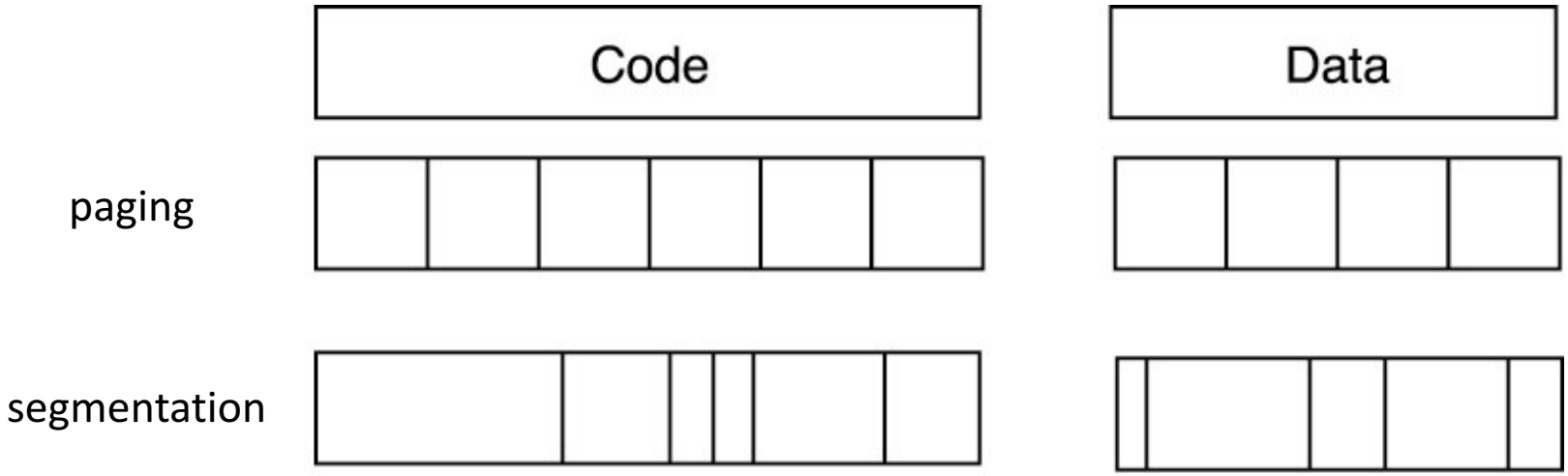
Allows each table to grow or shrink, independently

Comparison of Segmentation and Paging

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

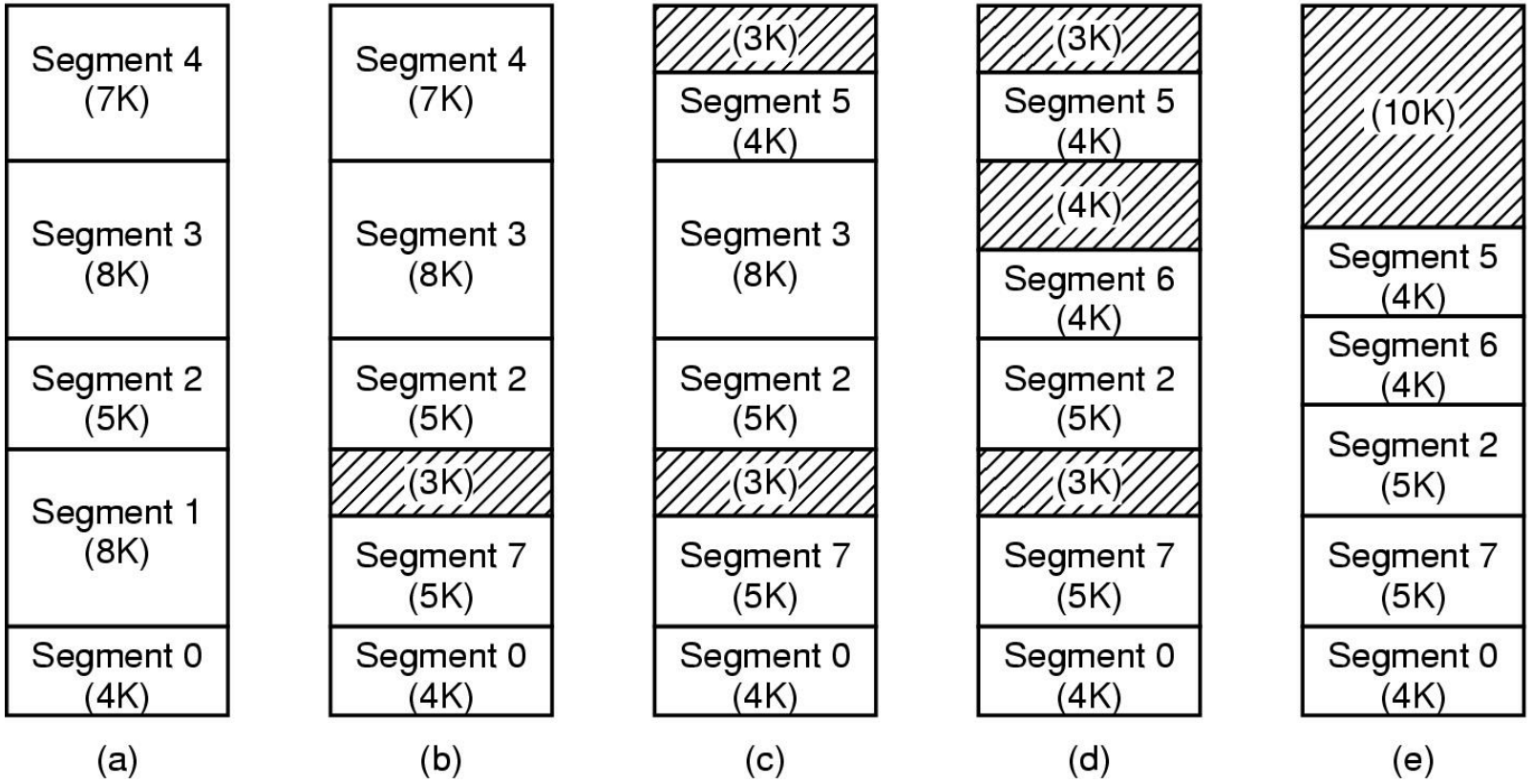
Paged v.s. Segmented Virtual Memory

- Paged virtual memory
  - Memory divided into fixed sized pages
- Segmented virtual memory
  - Memory divided into variable length segments



Implementation of Pure Segmentation

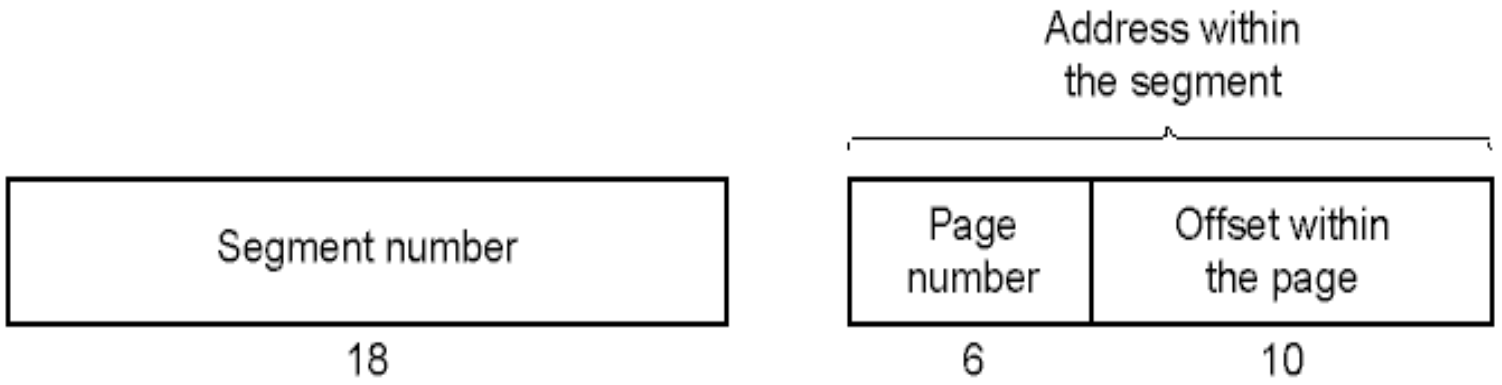
- An essential difference of paging and segmentation
  - Segments have different sizes while pages are fixed size!



(a)-(d) Development of checkerboarding (*external fragmentation*) in physical memory, if segments are small; (e) Removal of the checkerboarding by *compaction*

Segmentation with Paging: MULTICS

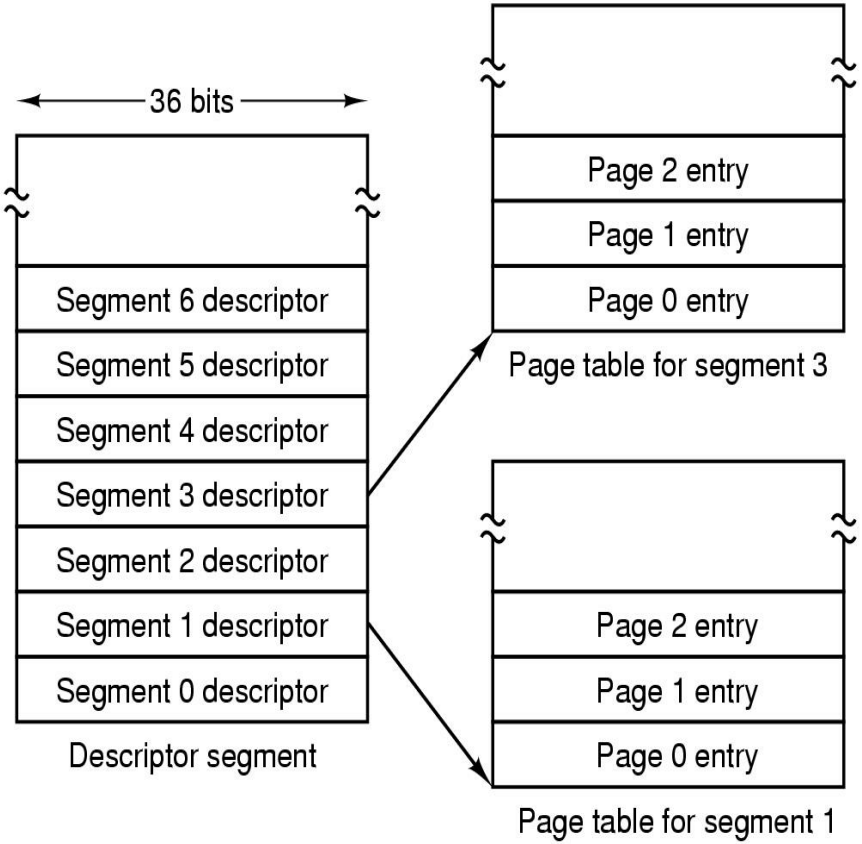
- What if the memory is not large enough for a single segment?
- MULTICS (Honeywell 6000)
  - Paged segment with word (4B) addressing
  - *Multi-dim* VM up to 2^18 segments, each up to 64K (32-bit) words
  - 34-bit virtual address (seg #, page #, page offset)
  - Physical memory 16M words (24-bit physical address)
  - Page size: 1024 words; or 64 words (64-word alignment)

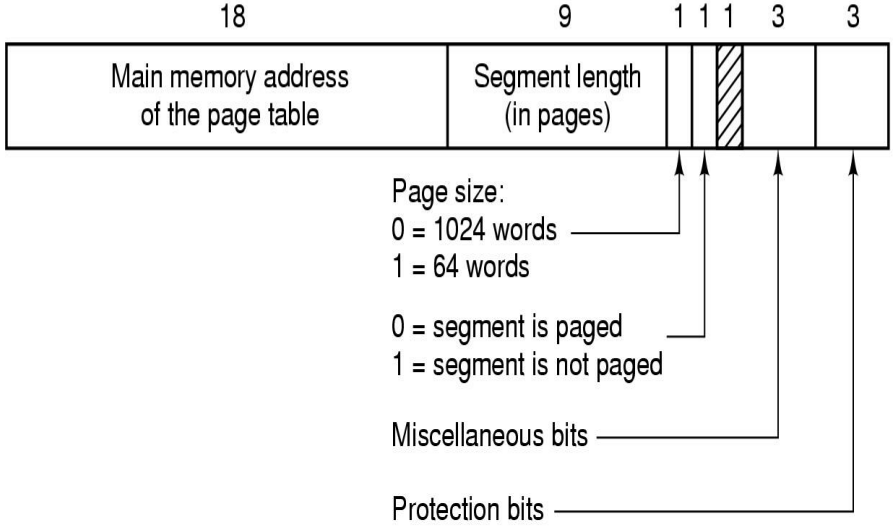


A 34-bit MULTICS virtual address

MULTICS Virtual Memory

- One descriptor segment and 36-bit segment descriptors
  - What if the page table of a segment is not in the memory?

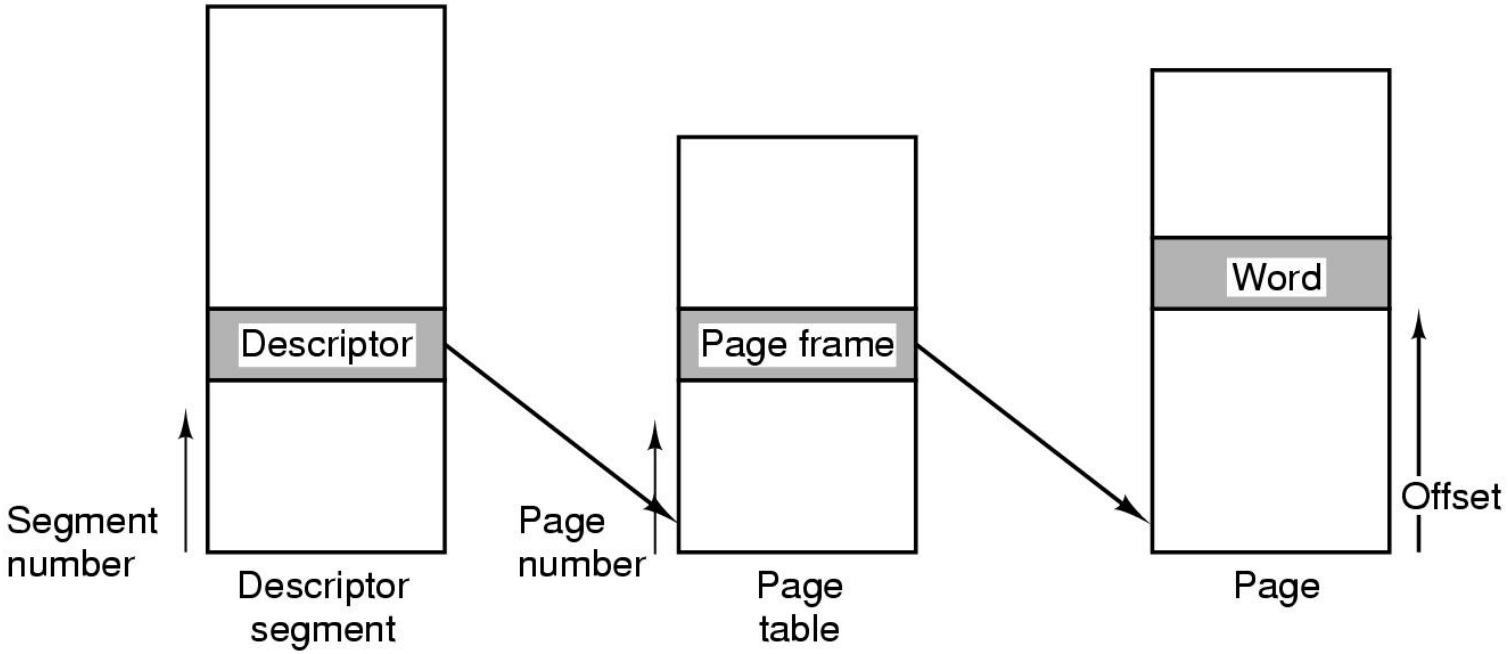




Descriptor segment points to page tables. Segment descriptor.

MULTICS Virtual Address → Physical Address

What if the descriptor segment is paged (often it is)?
How to speed up the searching &conversion?



Conversion of a 2-part MULTICS address into a main memory address

MULTICS TLB

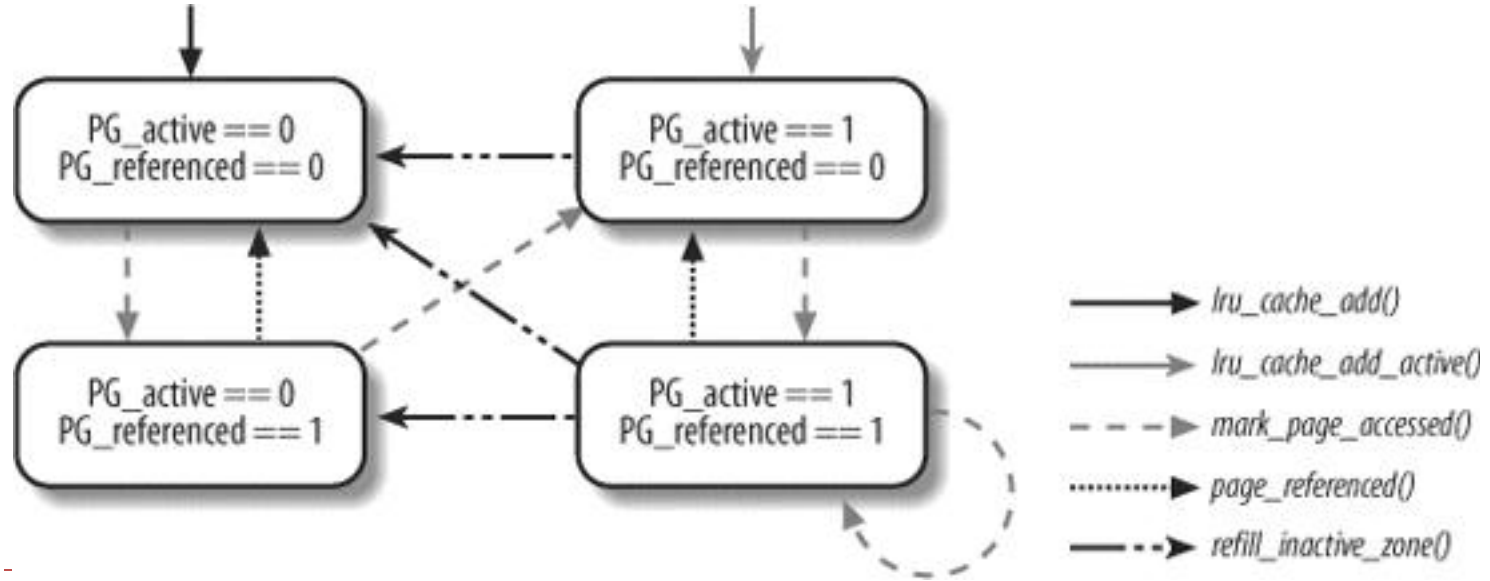
Comparison field: Segment number	Comparison field: Virtual page	Page frame	Protection	Age	Is this entry used?
4	1	7	Read/write	13	1
6	0	2	Read only	10	1
12	3	1	Read/write	2	1
					0
2	1	0	Execute only	7	1
2	2	12	Execute only	9	1

Simplified version of the MULTICS TLB (LRU replacement), which has 16 most recently referenced pages.

Case Study: Linux Page Replacement Algorithm

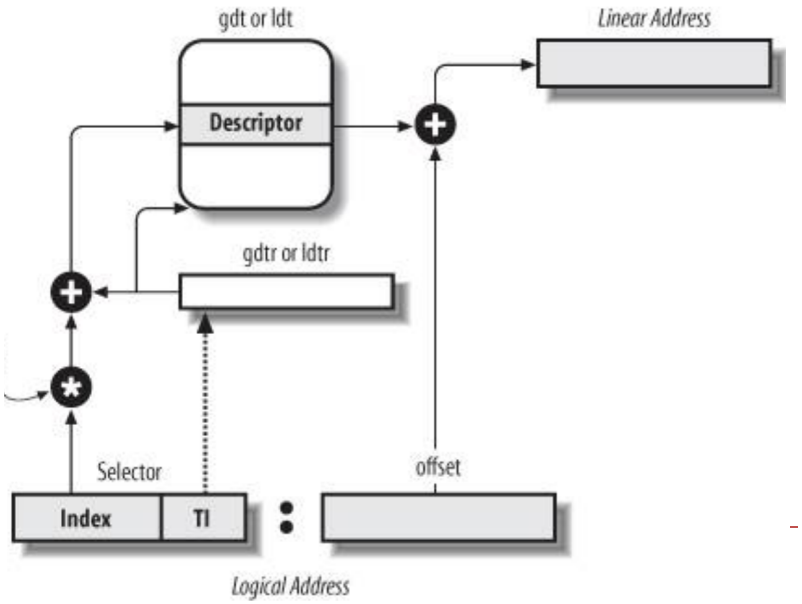
- Page Frame Reclaiming Algorithm (PFRA)
  - Unreclaimable
  - Swappable
  - Syncable
  - discardable

`SRC/include/linux/mm_types.h: page`

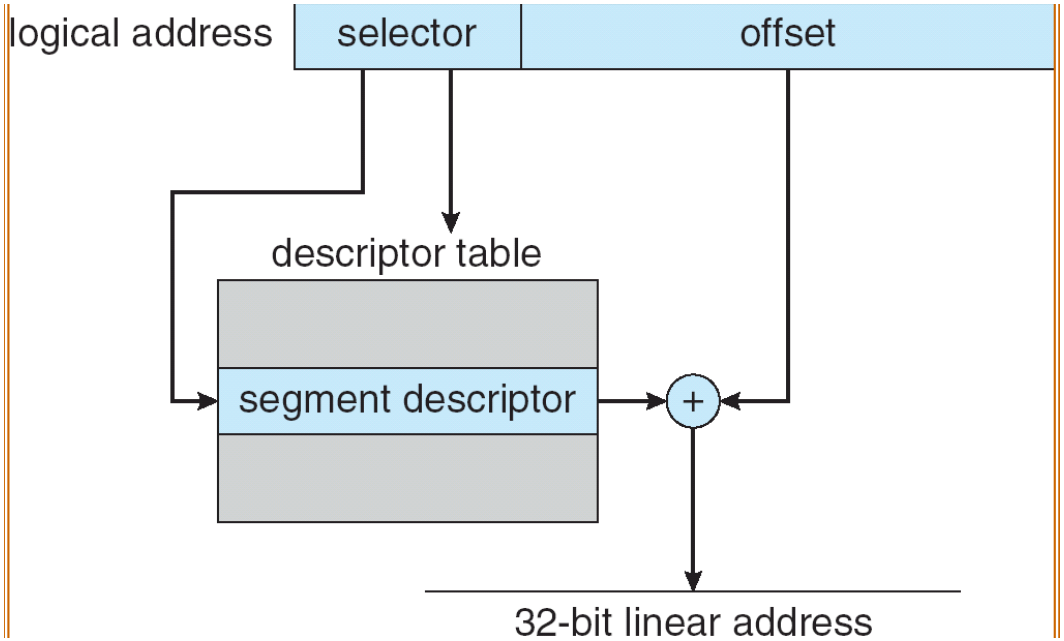


Case Study: Segmentation in Linux and X86_32

- Linux uses segmentation in a limited way
  - Paging is simple and effective
  - Portability to other architecture, e.g., RISC



Segmentation in X86_32 Hardware



Summary

- Page replacement algorithms
  - Comparison?
- Design issues
- Implementation issues
- Segmentation
  - Why?
  - Linux
- Self-reading
  - Instruction Backup
  - Locking Pages in Memory
  - Separation of Policy and Mechanism
  - Segmentation with Paging: Pentium