Sistemas Operativos

Licenciatura em Engenharia Informática
Licenciatura em Engenharia Computacional
Licenciatura em Física (opcional)

Ano letivo 2025/2026

Pedro Azevedo Fernandes (paf@ua.pt)

Memory Management and Virtual Memory

Disclamer

- The slides are authored by Andrew Tanenbaum

Memory

- Paraphrase of Parkinson's Law, "Programs expand to fill the memory available to hold them."
- Average home computer nowadays has 10,000 times more memory than the IBM 7094, the largest computer in the world in the early 1960s

No Memory Abstraction

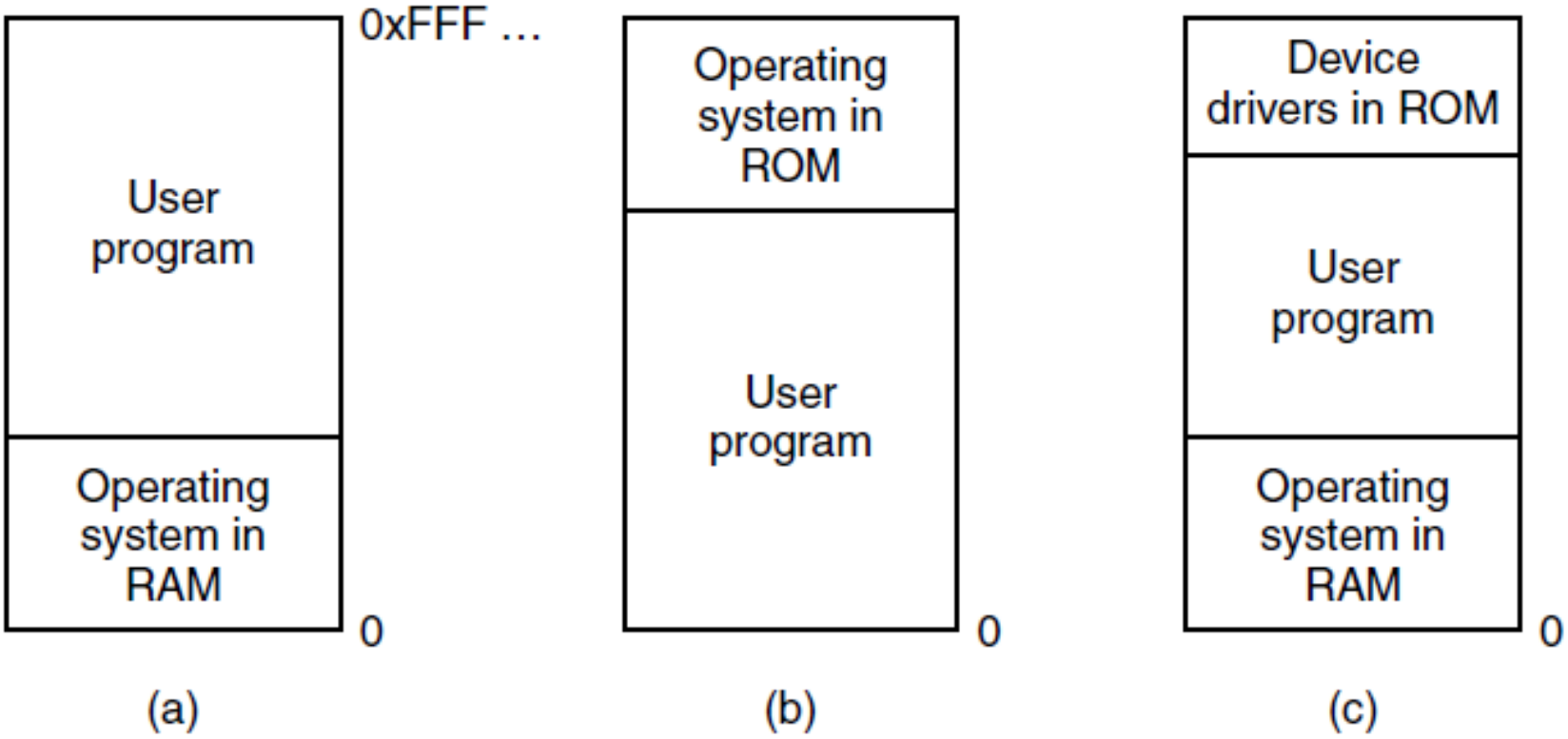


Figure 3-1. Three simple ways of organizing memory with an operating system and one user process. Other possibilities also exist

Running Multiple Programs Without a Memory Abstraction

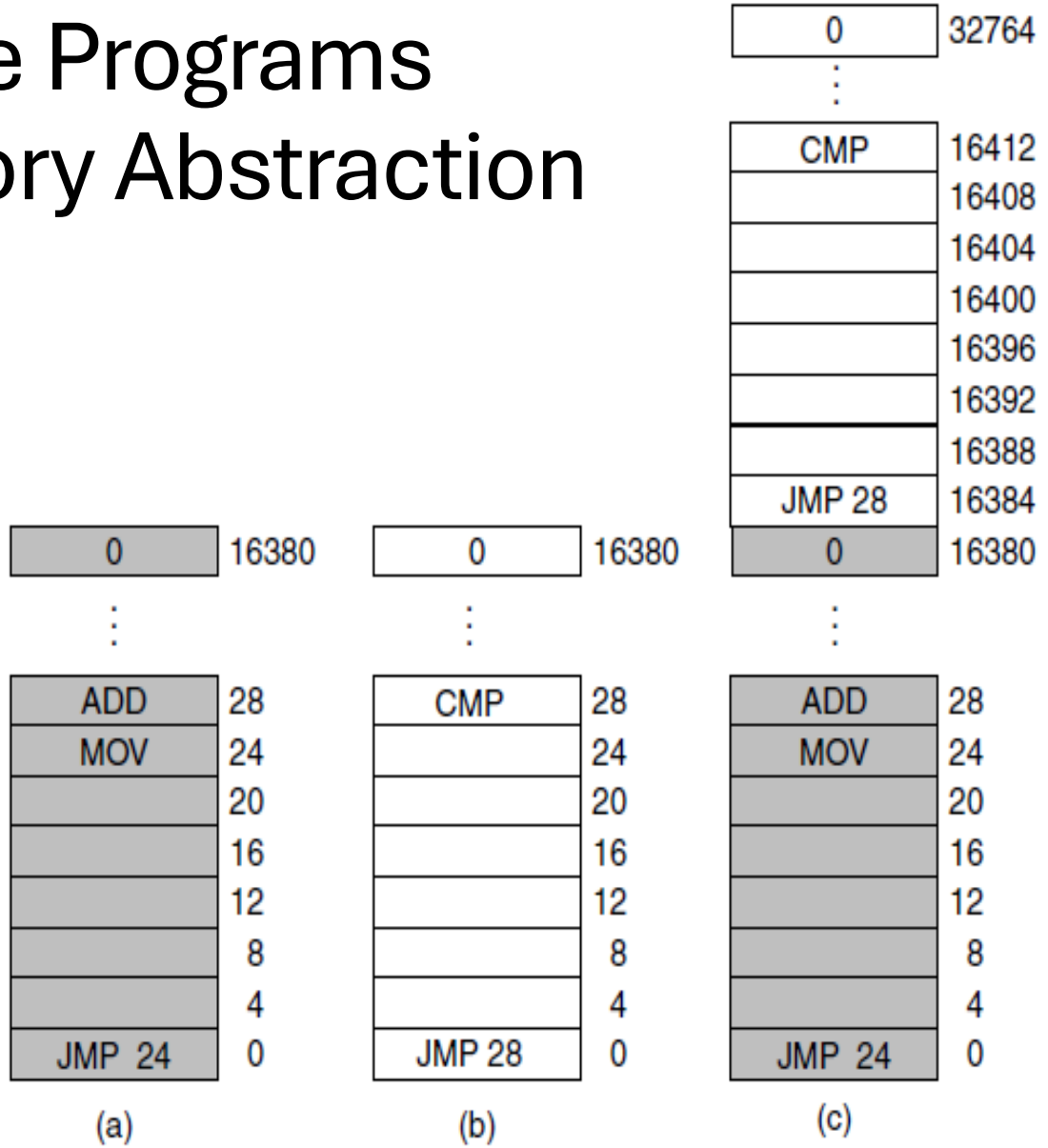


Figure 3-2. Illustration of the relocation problem. (a) A 16-KB program. (b) Another 16-KB program. (c) The two programs loaded consecutively into memory

Memory Abstraction

The address space

- the set of addresses that a process can use to address memory
- Each process has its own address space, independent of those belonging to other processes

Base and Limit Registers

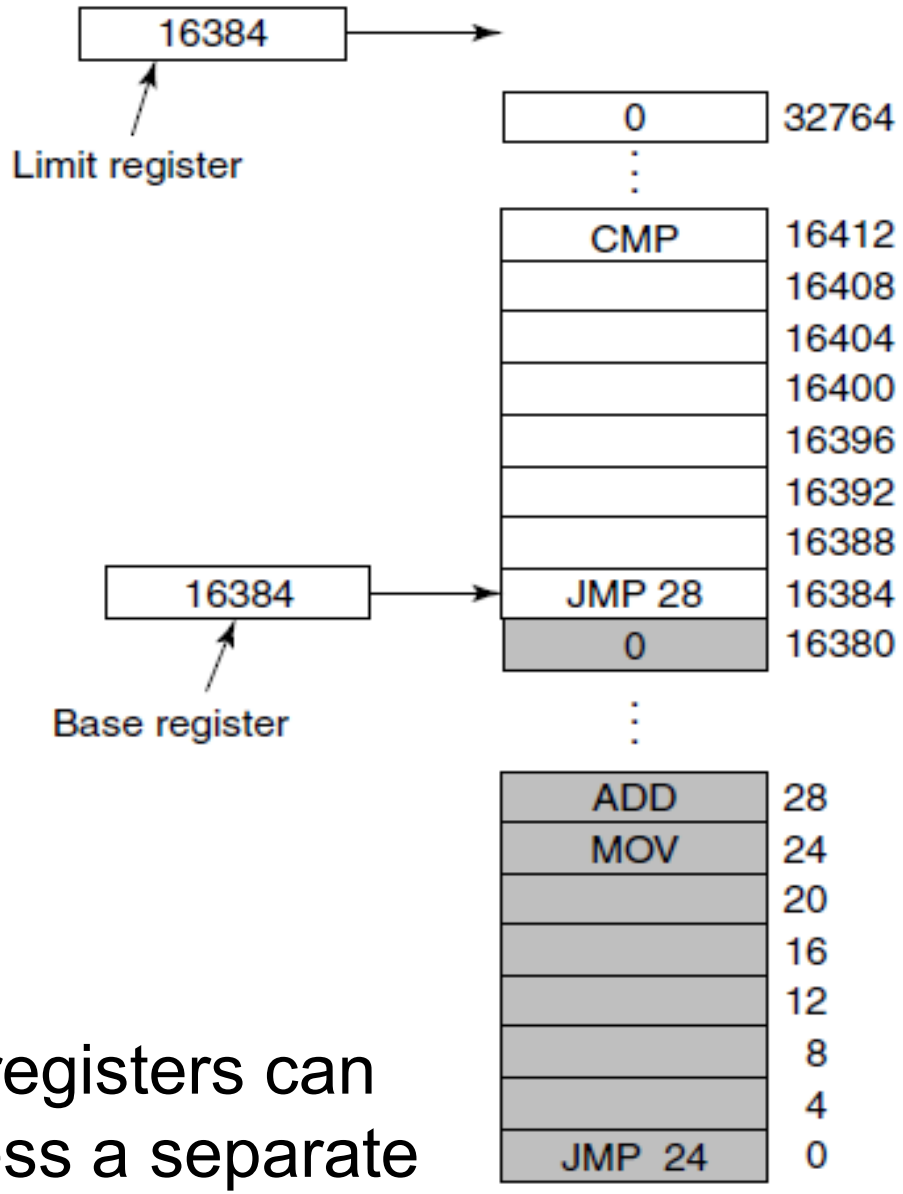


Figure 3-3. Base and limit registers can be used to give each process a separate address space.

Swapping (1)

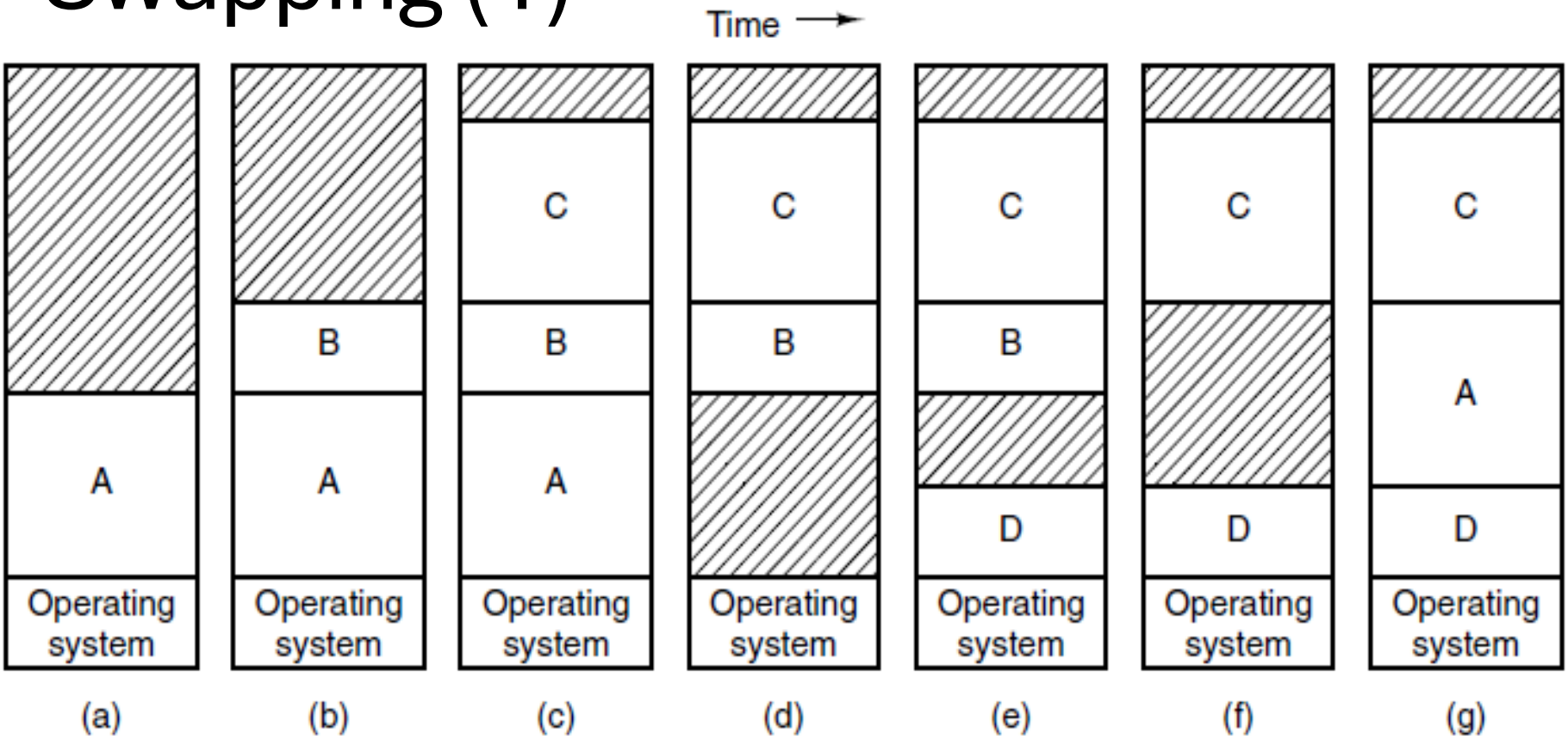


Figure 3-4. Memory allocation changes as processes come into memory and leave it. The shaded regions are unused memory

Swap-ping (2)

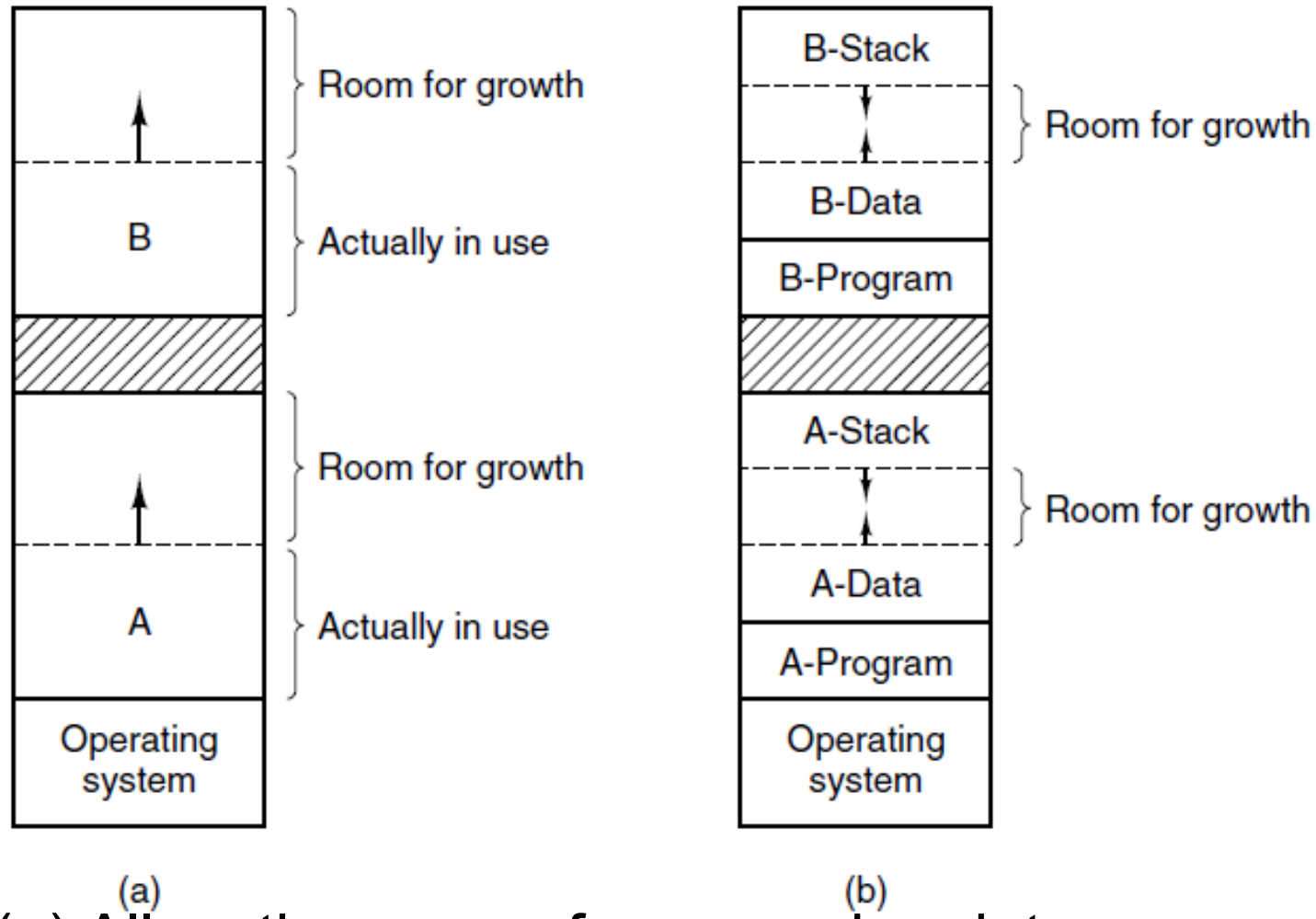


Figure 3-5. (a) Allocating space for a growing data segment. (b) Allocating space for a growing stack and a growing data segment.

Memory Management with Bitmaps

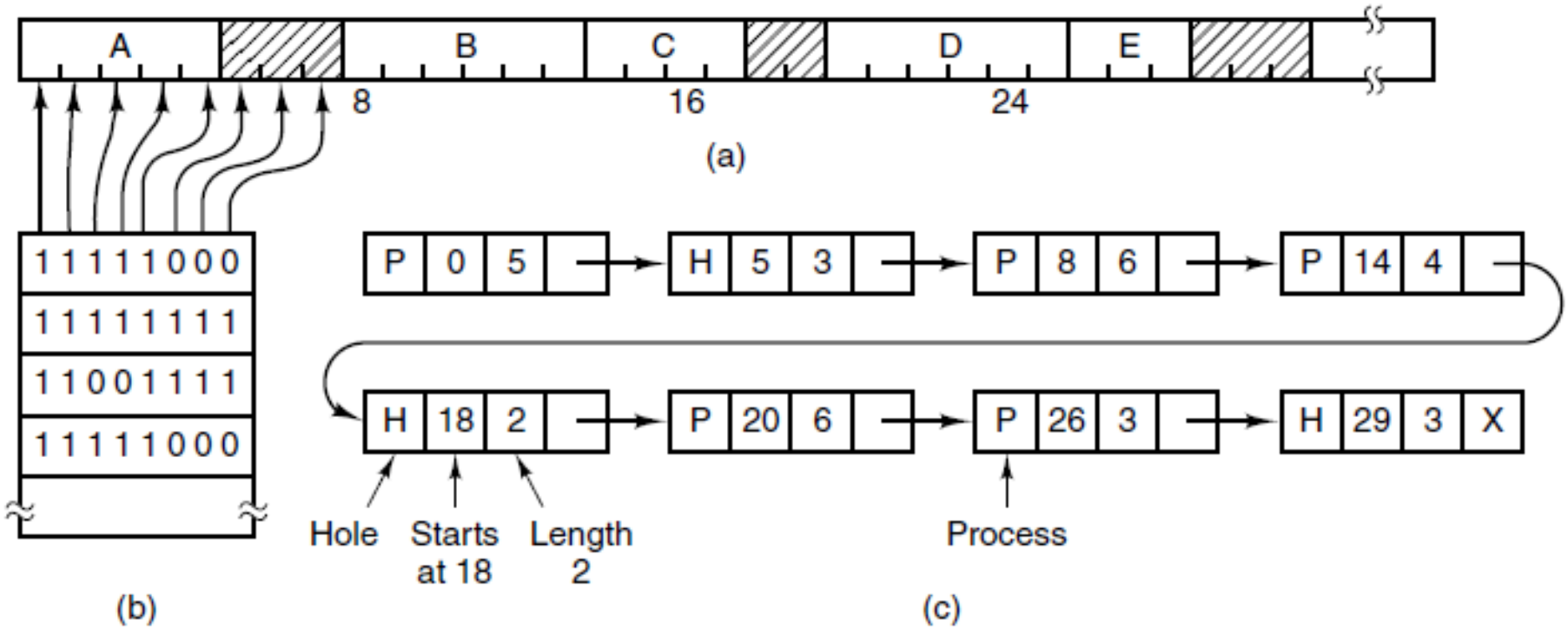


Figure 3-6. (a) A part of memory with five processes and three holes. The tickmarks show the memory allocation units. The shaded regions (0 in the bitmap) are free. (b) The corresponding bitmap. (c) The same information as a list.

Memory Management with Linked Lists

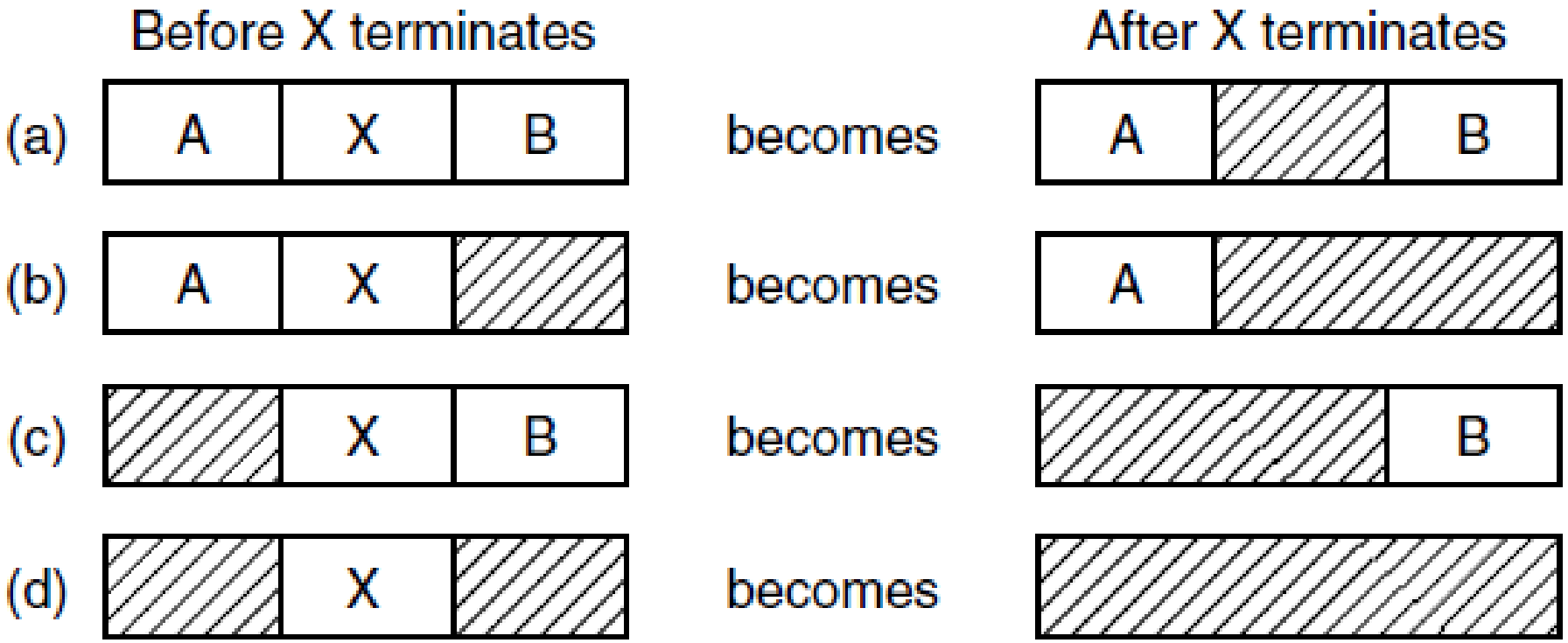


Figure 3-7. Four neighbor combinations for the terminating process, X.

Memory Management Algorithms

- First fit
- Next fit
- Best fit
- Worst fit
- Quick fit

Virtual Memory

- There is a need to run programs that are too large to fit in memory
- Solution adopted in the 1960s, split programs into little pieces, called overlays
  - Kept on the disk, swapped in and out of memory
- Virtual memory : each program has its own address space, broken up into chunks called pages

Paging (1)

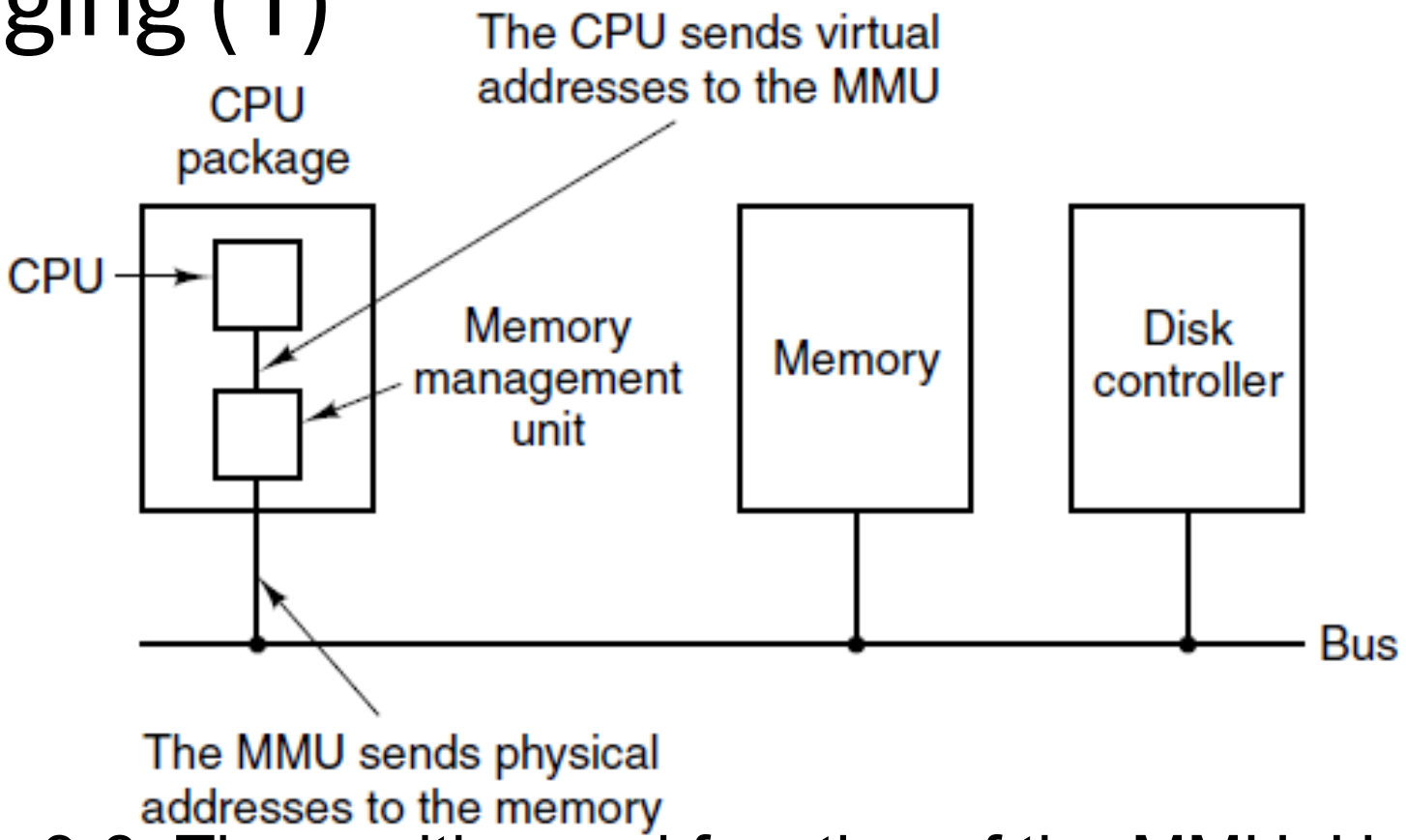
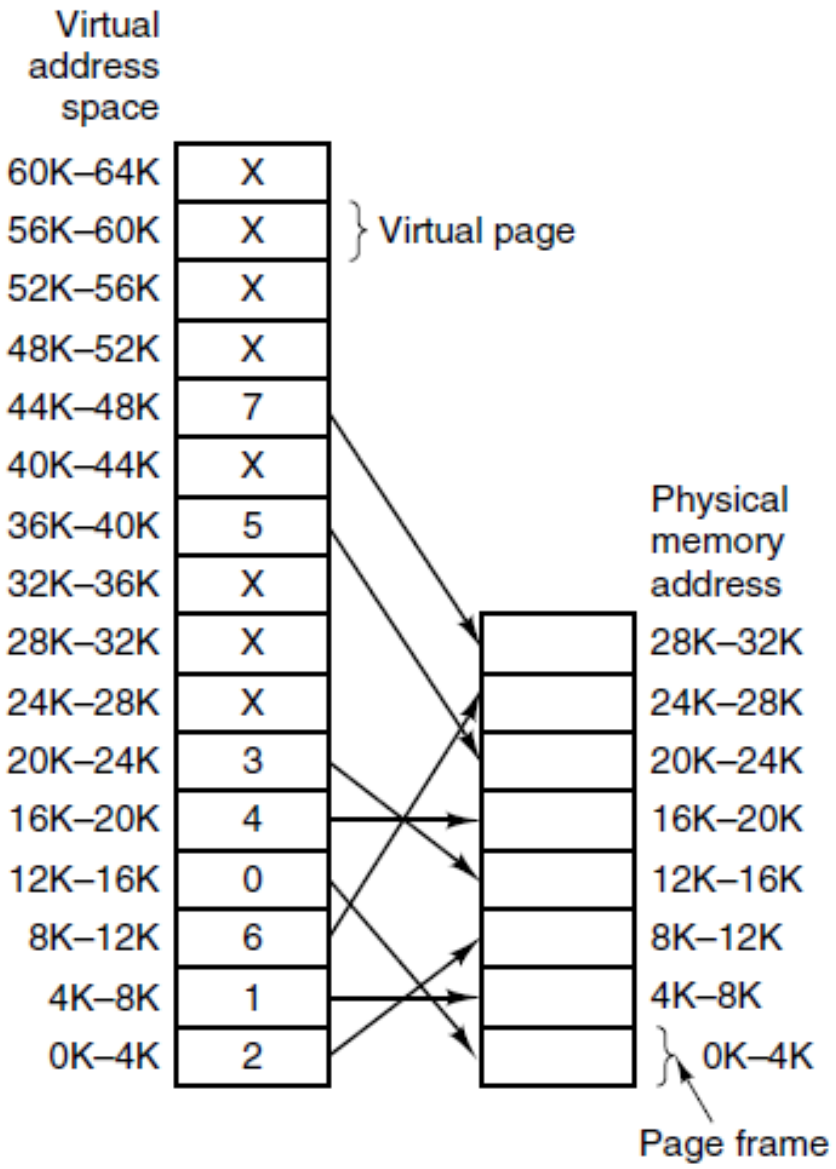


Figure 3-8. The position and function of the MMU. Here the MMU is shown as being a part of the CPU chip because it commonly is nowadays. However, logically it could be a separate chip and was years ago.

Paging (2)

Figure 3-9. The relation between virtual addresses and physical memory addresses is given by the page table. Every page begins on a multiple of 4096 and ends 4095 addresses higher, so 4K–8K really means 4096–8191 and 8K to 12K means 8192–12287



Paging (3)

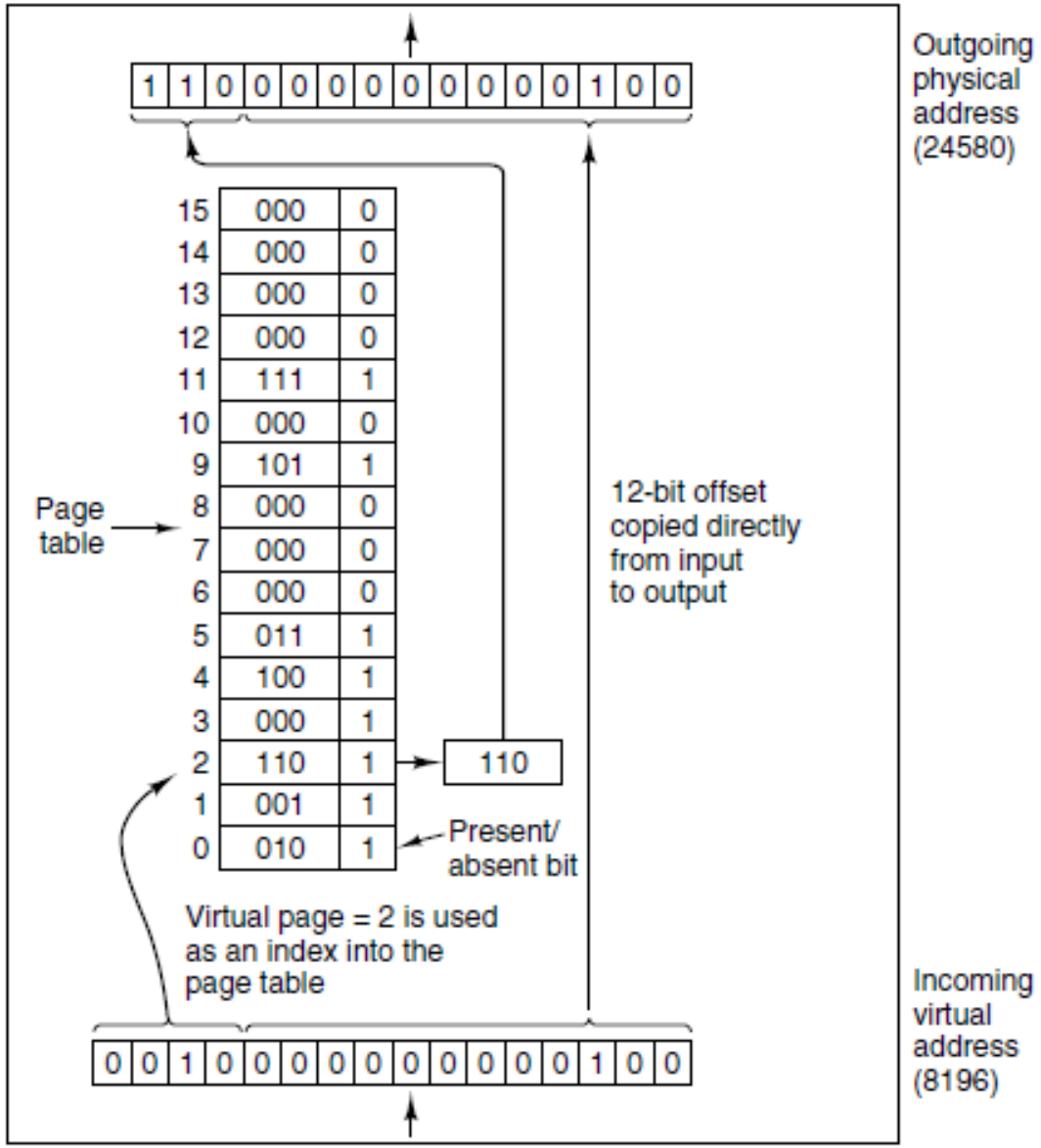


Figure 3-10. The internal operation of the MMU with 16 4-KB pages.

Structure of a Page Table Entry

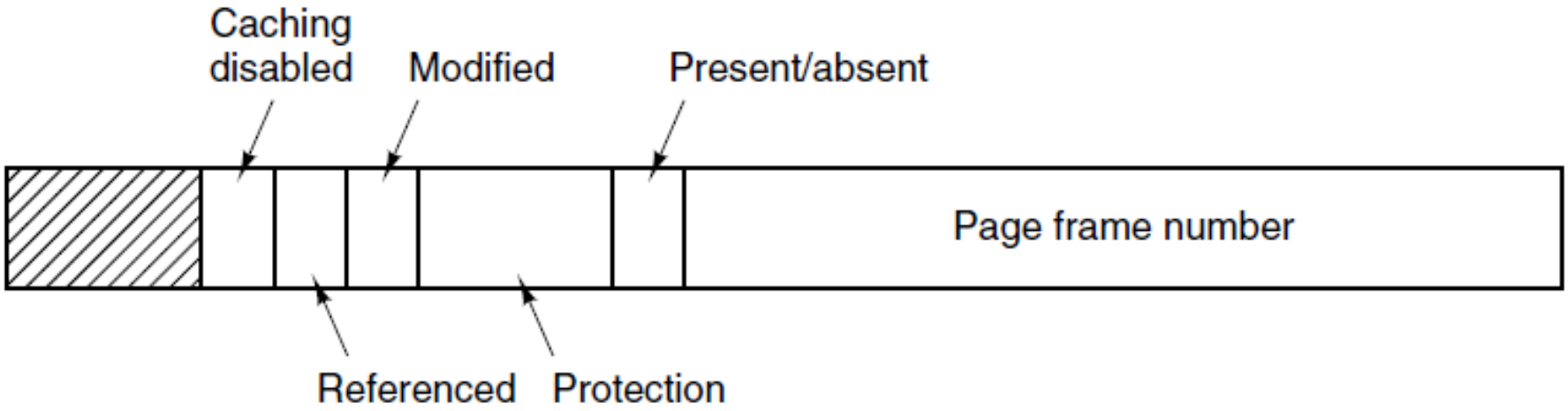


Figure 3-11. A typical page table entry

Speeding Up Paging

- Major issues faced:
- The mapping from virtual address to physical address must be fast.
- If the virtual address space is large, the page table will be large.

Translation Lookaside Buffers

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

Figure 3-12. A TLB to speed up paging

Multilevel Page Tables

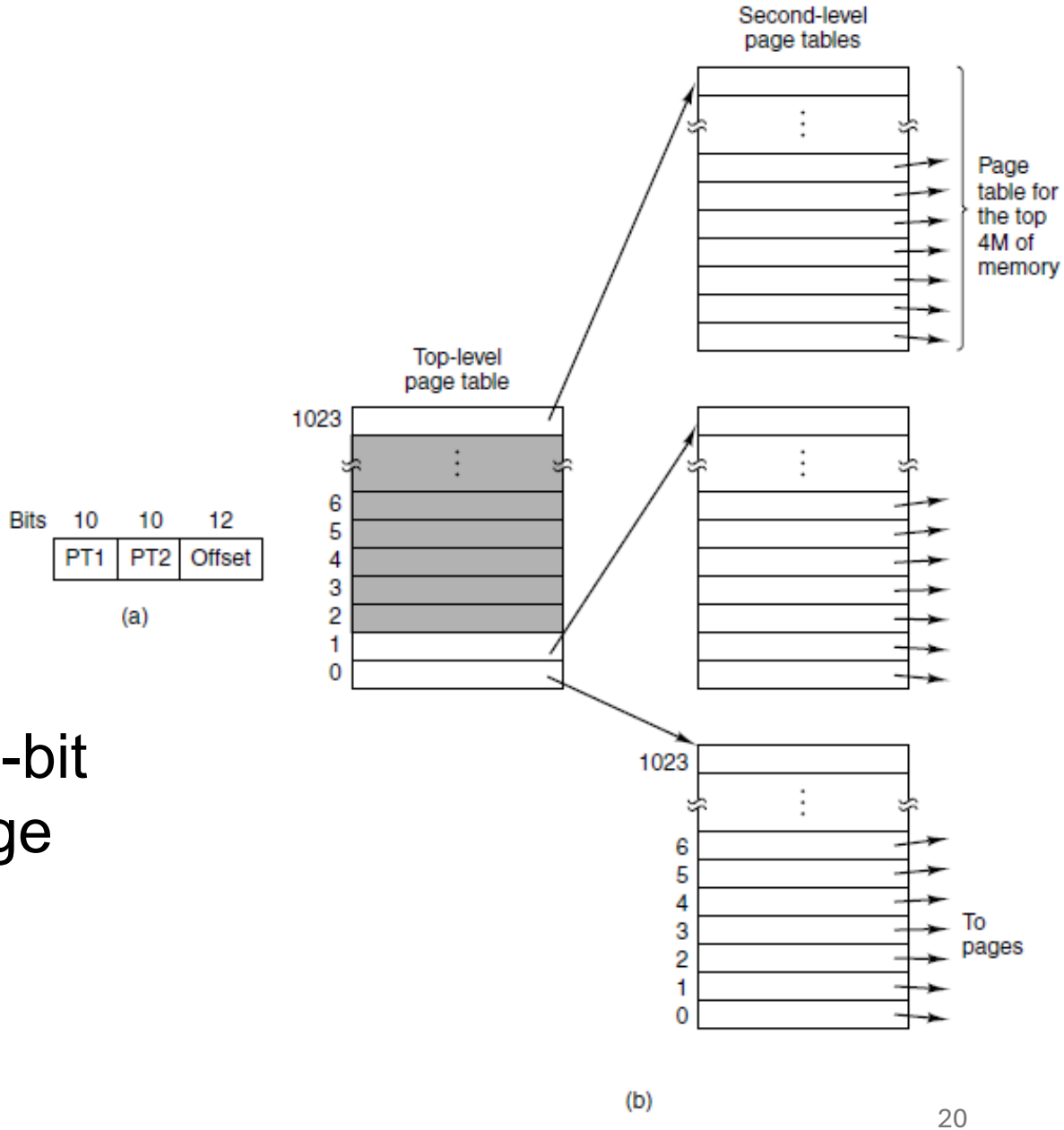


Figure 3-13. (a) A 32-bit address with two page table fields. (b) Two-level page tables.

Inverted Page Tables

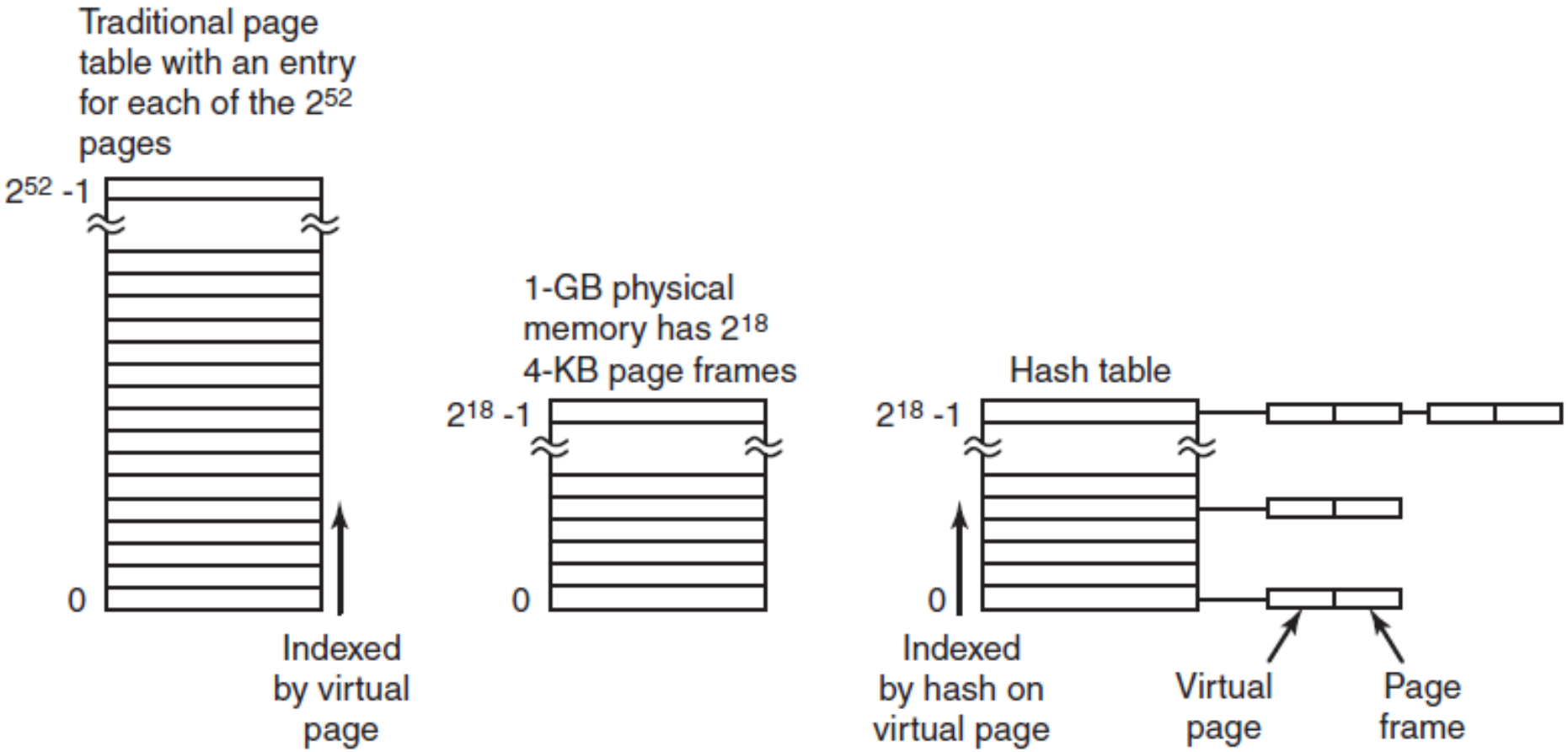


Figure 3-14. Comparison of a traditional page table with an inverted page table.

Page Replacement Algorithms

- Optimal algorithm
- Not recently used algorithm
- First-in, first-out (FIFO) algorithm
- Second-chance algorithm
- Clock algorithm
- Least recently used (LRU) algorithm
- Working set algorithm
- WSClock algorithm

Not Recently Used Algorithm

- At page fault, system inspects pages
- Categories of pages based on the current values of their R and M bits:
  - Class 0: not referenced, not modified.
  - Class 1: not referenced, modified.
  - Class 2: referenced, not modified.
  - Class 3: referenced, modified.

Second-Chance Algorithm

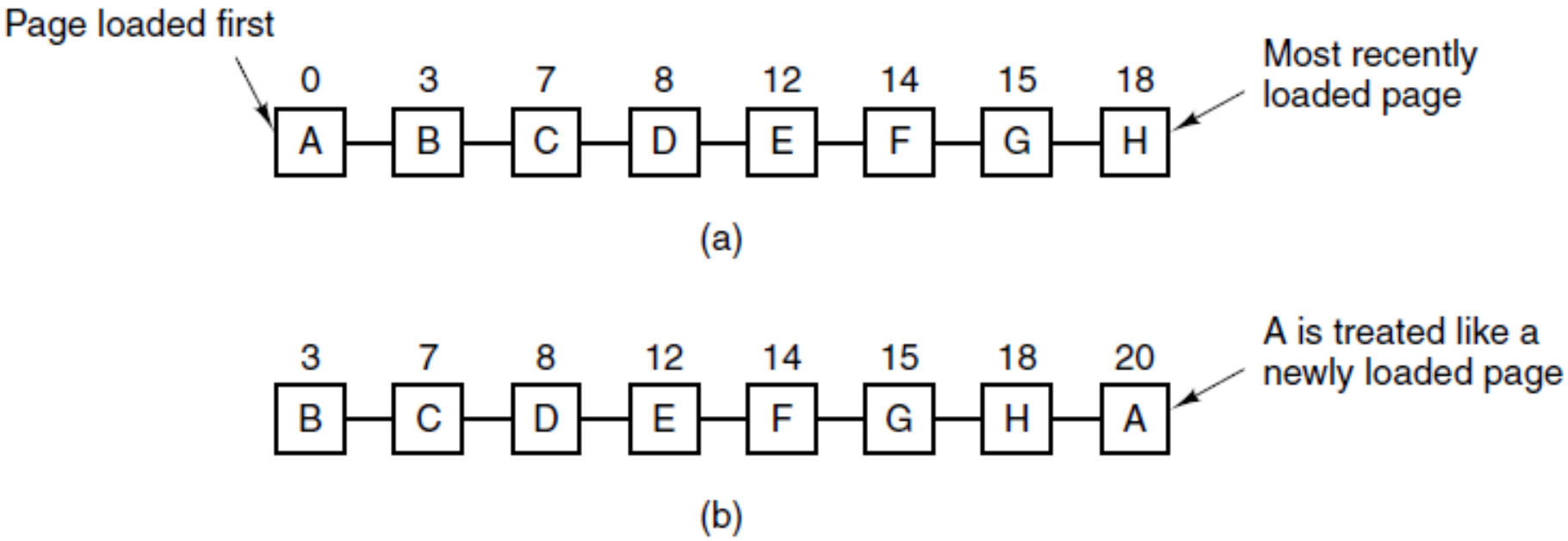
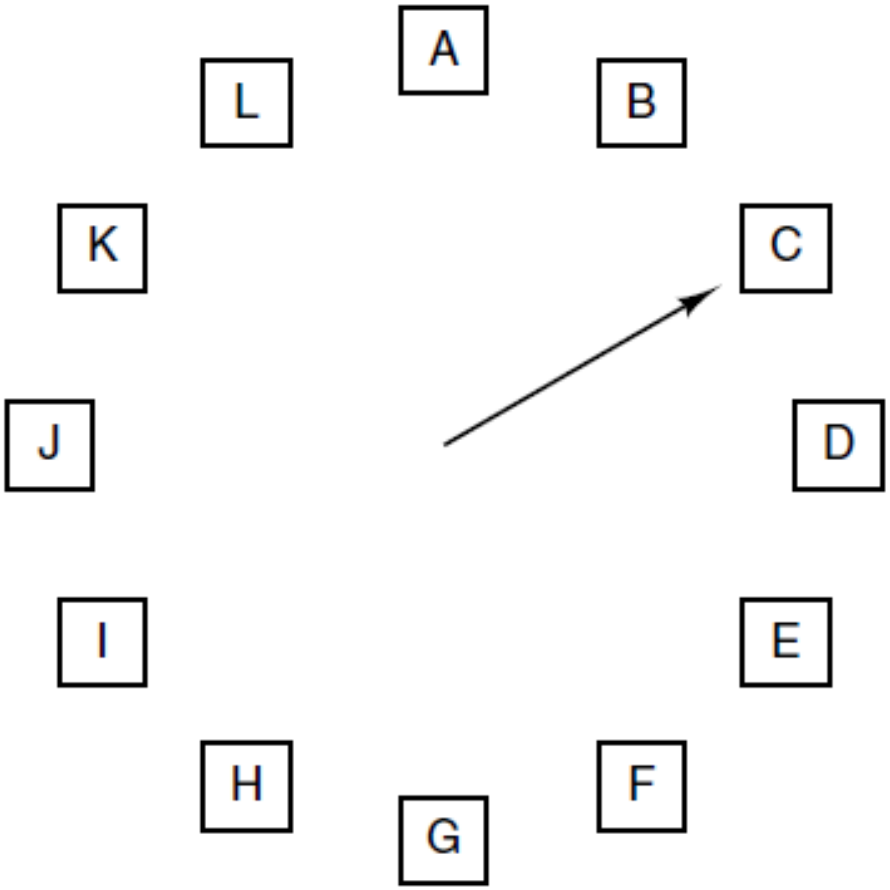


Figure 3-15. Operation of second chance. (a) Pages sorted in FIFO order. (b) Page list if a page fault occurs at time 20 and A has its R bit set. The numbers above the pages are their load times.

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

Figure 3-16. The clock page replacement algorithm.

Simulating LRU in Software

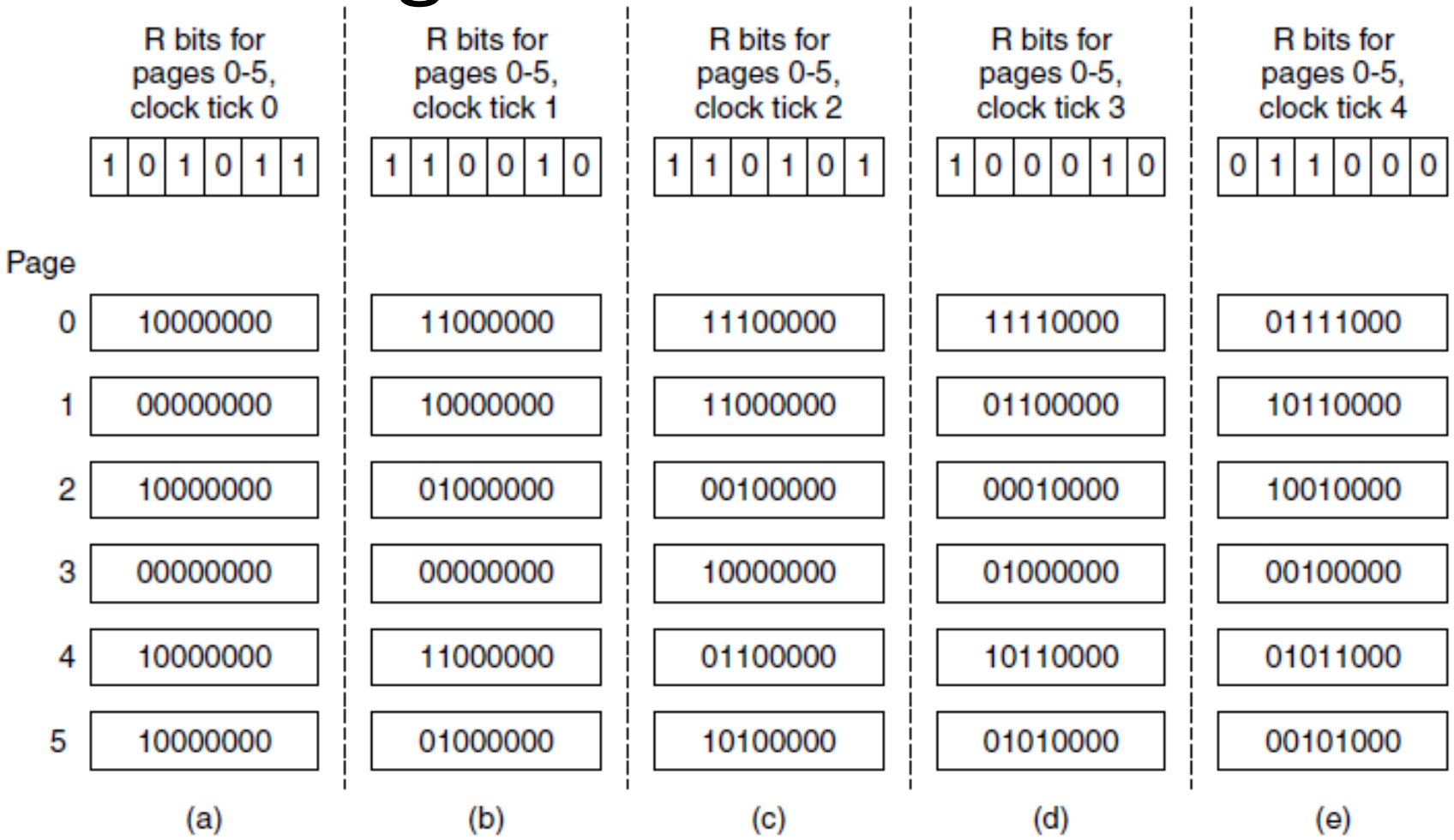


Figure 3-17. The aging algorithm simulates LRU in software. Shown are six pages for five clock ticks. The five clock ticks are represented by (a) to (e).

Working Set Algorithm (1)

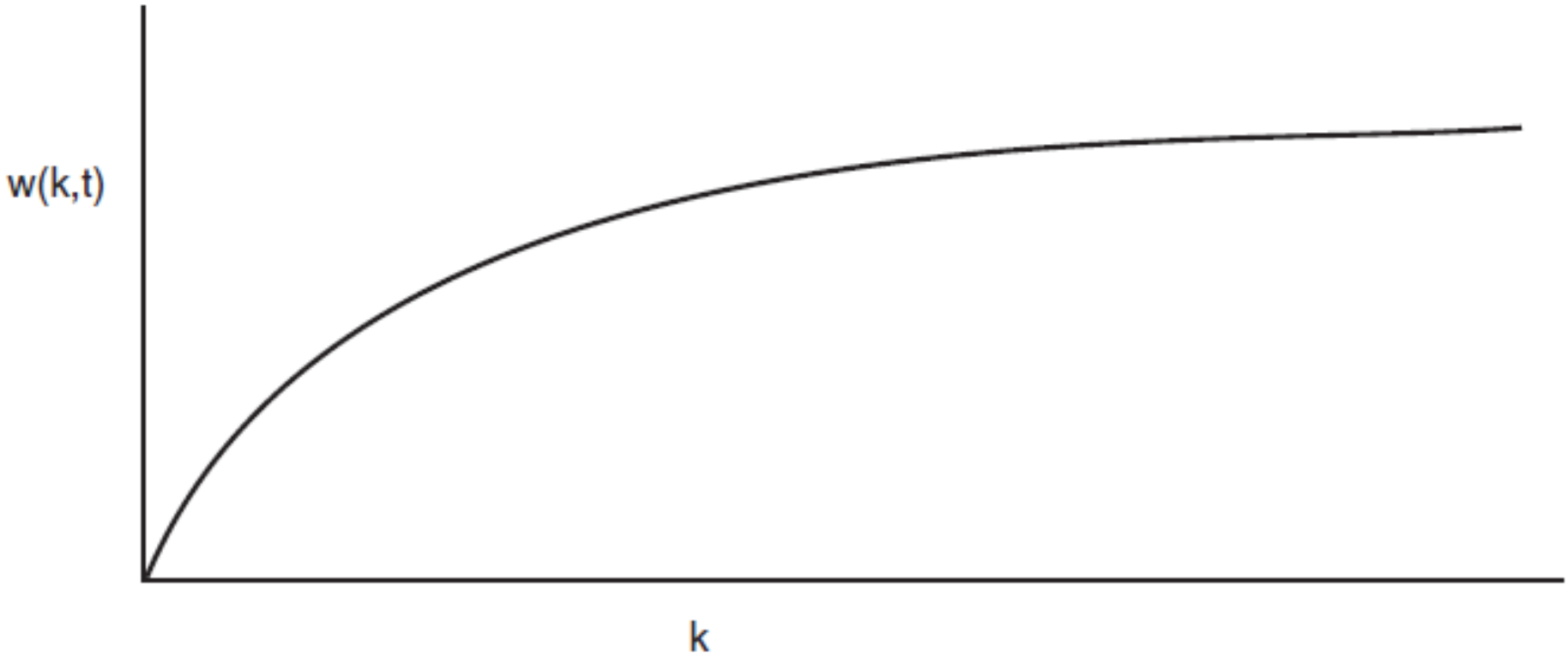


Figure 3-18. The working set is the set of pages used by the k most recent memory references. The function w(k, t) is the size of the working set at time t.

Working Set Algorithm (2)

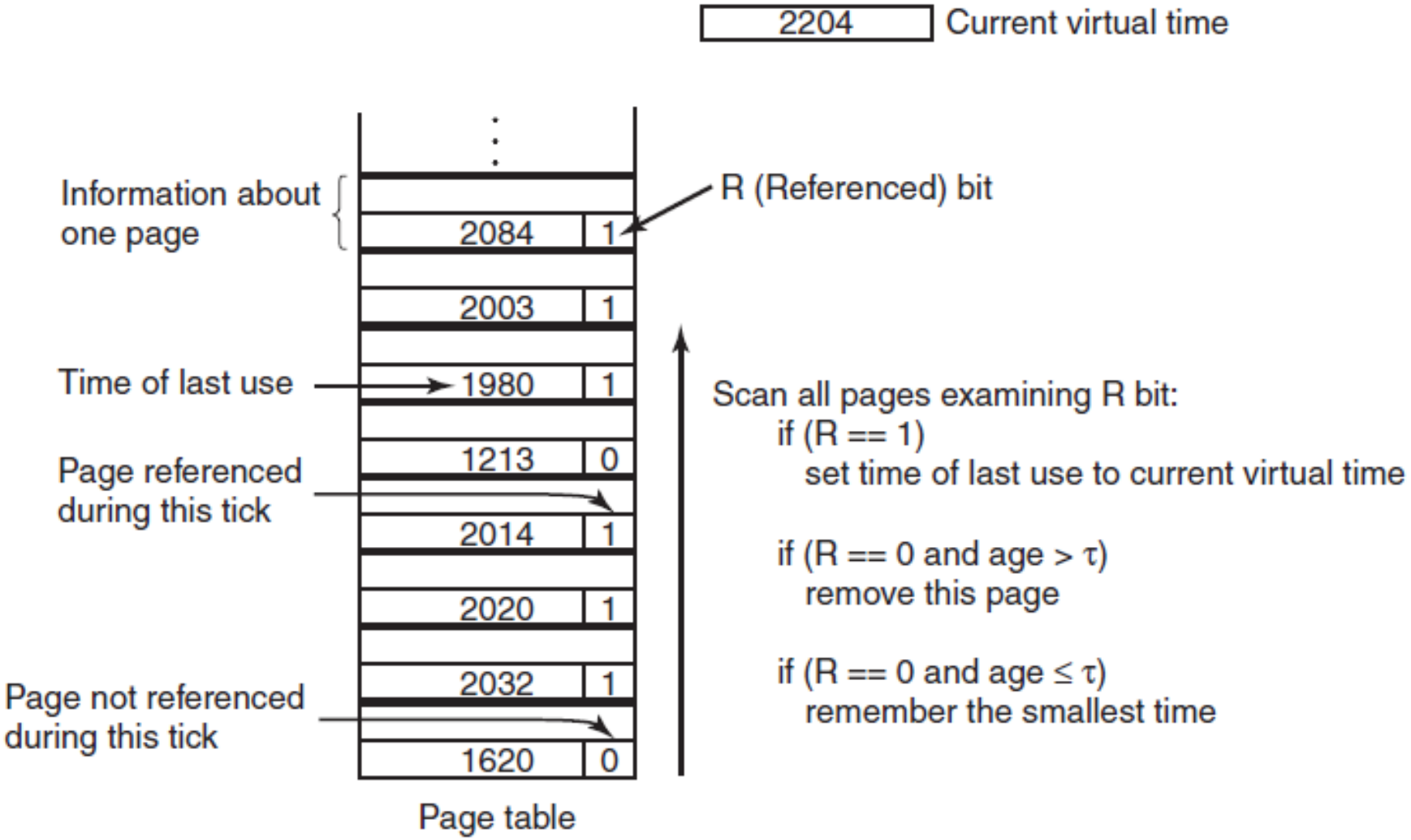


Figure 3-19. The working set algorithm.

WSClock Algorithm (1)

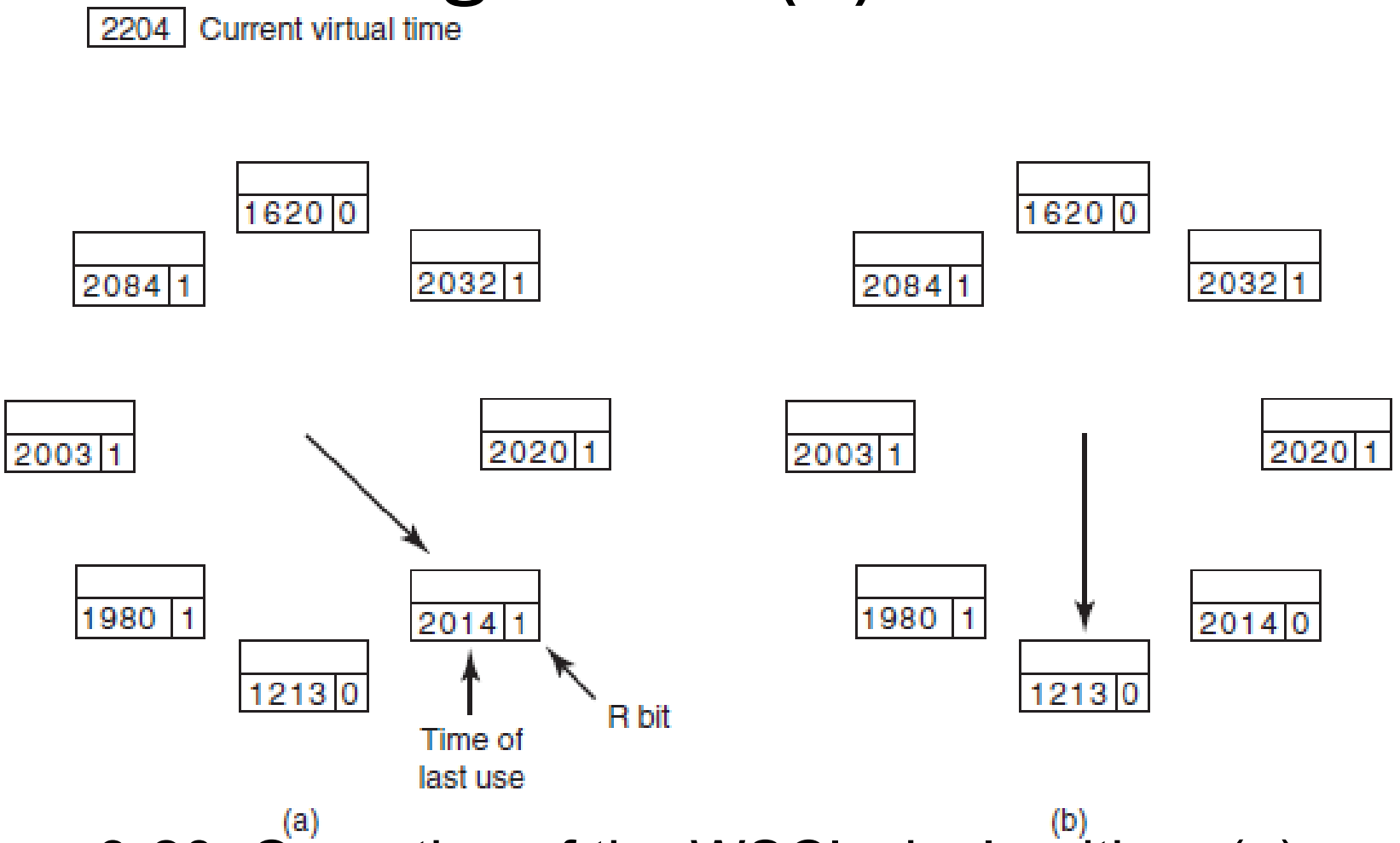


Figure 3-20. Operation of the WSClock algorithm. (a) and (b) give an example of what happens when R = 1.

WSClock Algorithm (2)

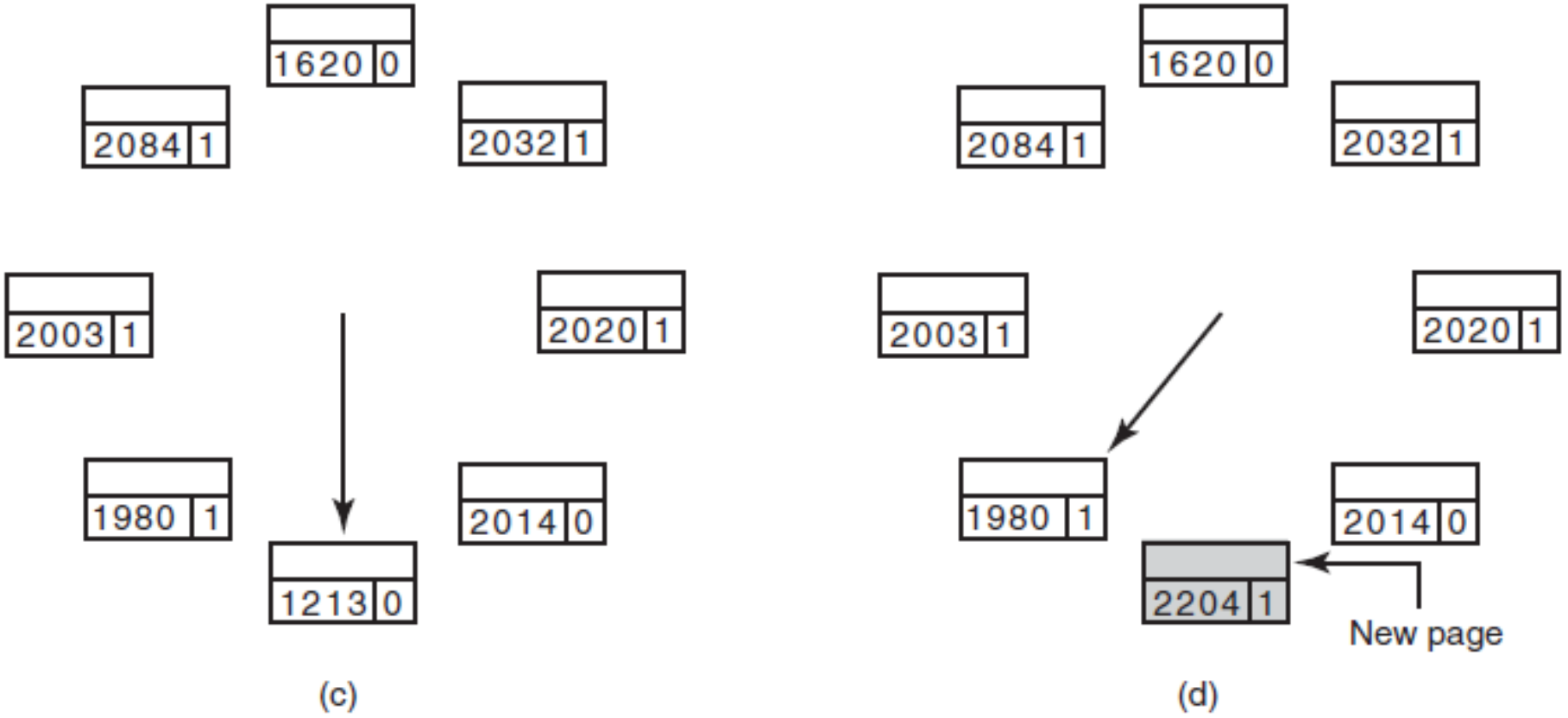


Figure 3-20. Operation of the WSClock algorithm. (c) and (d) give an example of R = 0.

Summary of Page Replacement Algorithms

Algorithm	Comment
Optimal	Not implementable, but useful as a benchmark
NRU (Not Recently Used)	Very crude approximation of LRU
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

Figure 3-21. Page replacement algorithms discussed in the text.

Local versus Global Allocation Policies (1)

	Age
A0	10
A1	7
A2	5
A3	4
A4	6
A5	3
B0	9
B1	4
B2	6
B3	2
B4	5
B5	6
B6	12
C1	3
C2	5
C3	6

(a)

A0
A1
A2
A3
A4
A6
B0
B1
B2
B3
B4
B5
B6
C1
C2
C3

(b)

A0
A1
A2
A3
A4
A5
B0
B1
B2
A6
B4
B5
B6
C1
C2
C3

(c)

Figure 3-22. Local versus global page replacement. (a) Original configuration. (b) Local page replacement. (c) Global page replacement.

Local versus Global Allocation Policies (2)

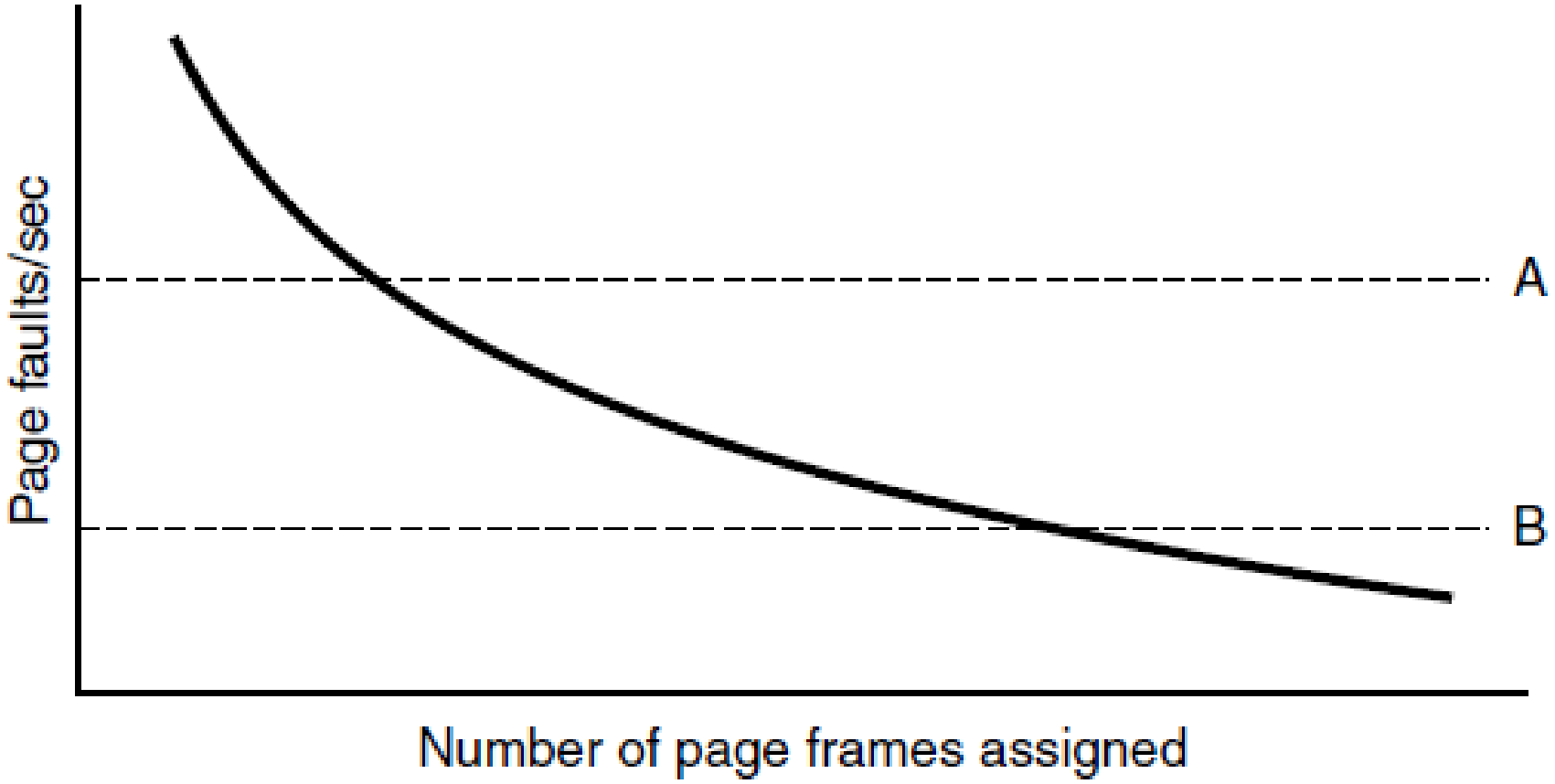


Figure 3-23. Page fault rate as a function of the number of page frames assigned.

Separate Instruction and Data Spaces

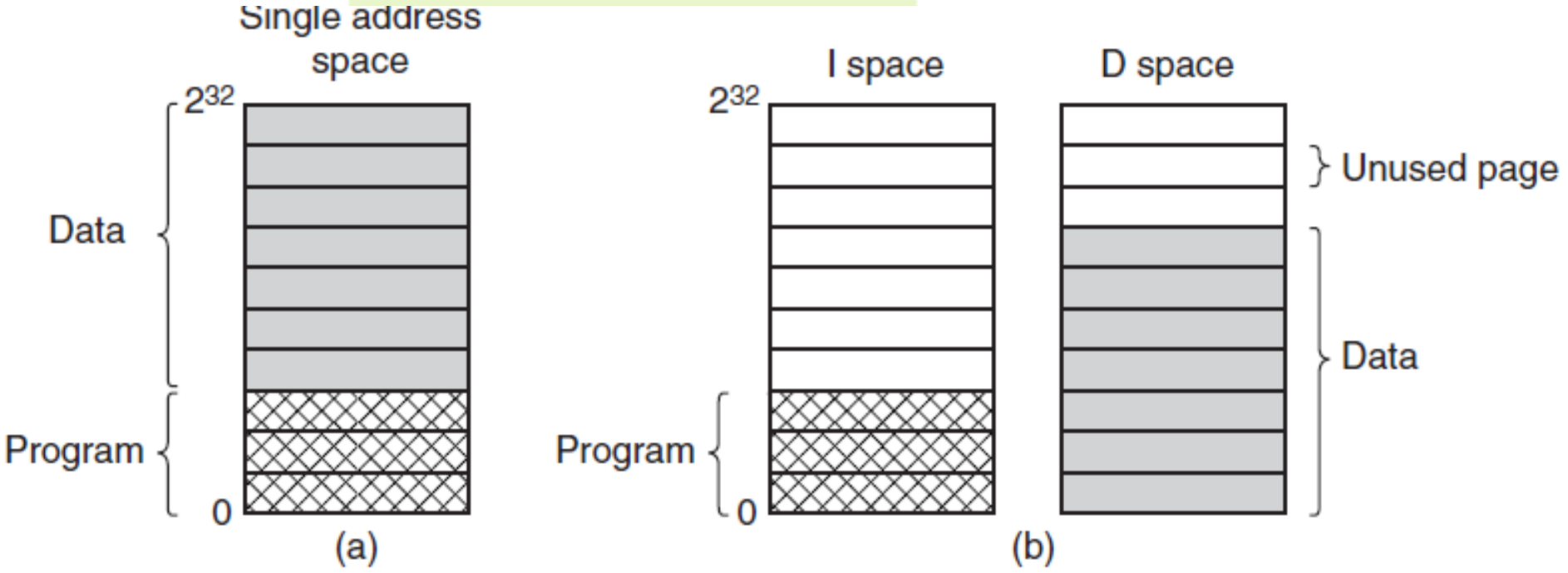


Figure 3-24. (a) One address space.
(b) Separate I and D spaces.

Shared Pages

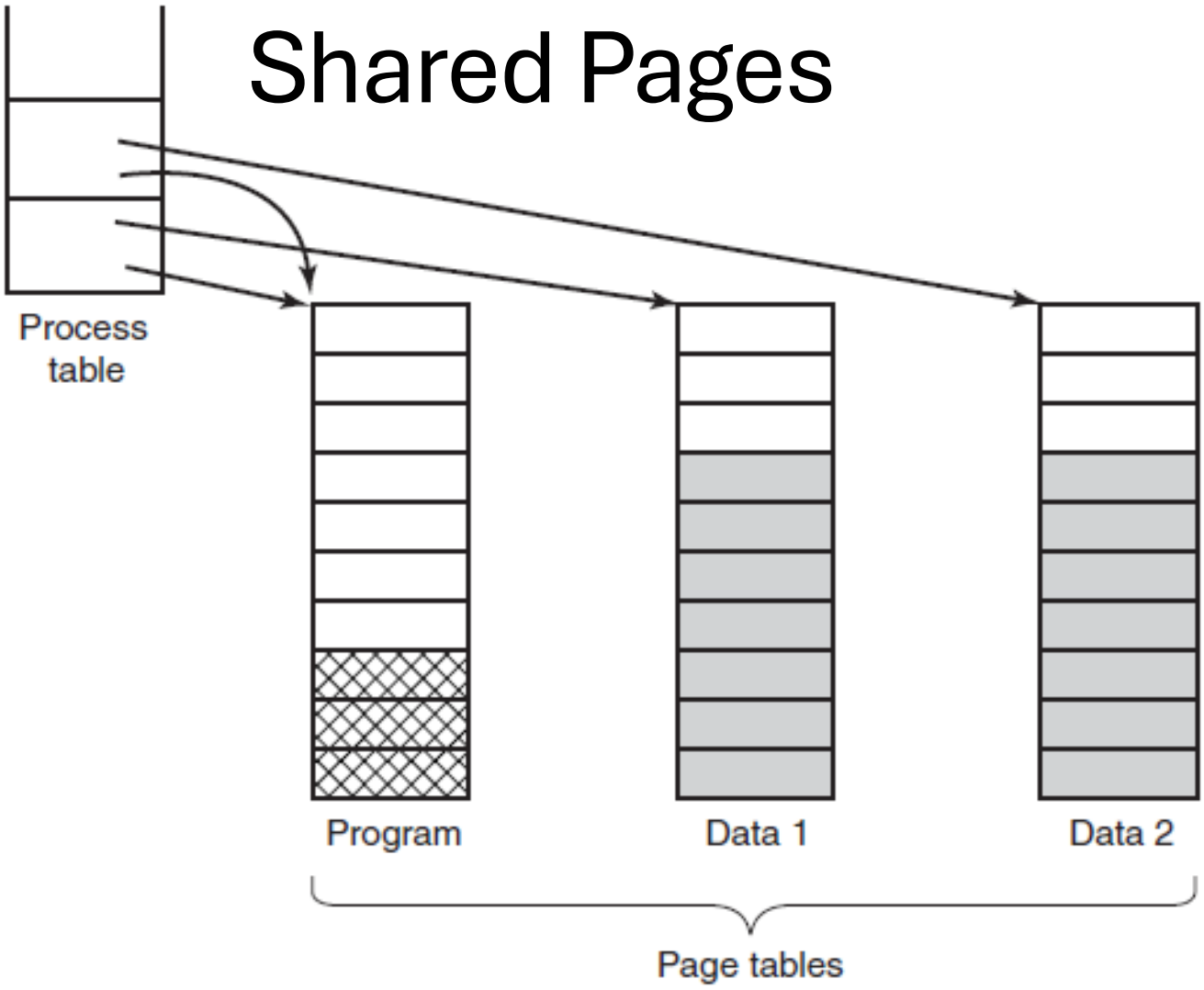


Figure 3-25. Two processes sharing the same program sharing its page table.

Shared Libraries

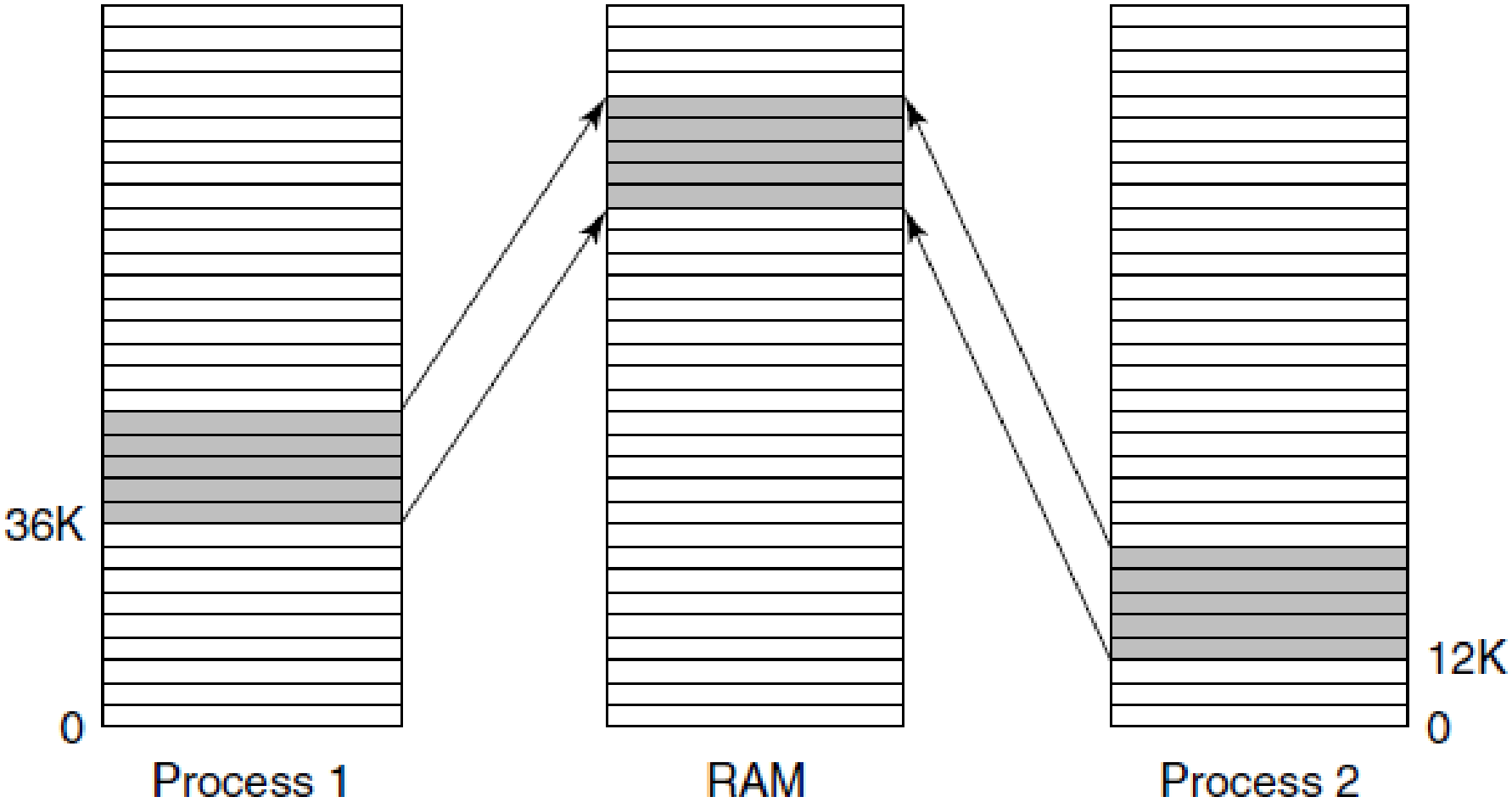


Figure 3-26. A shared library being used by two processes.

Page Fault Handling (1)

1. The hardware traps to kernel, saving program counter on stack.
2. Assembly code routine started to save general registers and other volatile info
3. system discovers page fault has occurred, tries to discover which virtual page needed
4. Once virtual address caused fault is known, system checks to see if address valid and the protection consistent with access

Page Fault Handling (2)

5. If frame selected dirty, page is scheduled for transfer to disk, context switch takes place, suspending faulting process
6. As soon as frame clean, operating system looks up disk address where needed page is, schedules disk operation to bring it in.
7. When disk interrupt indicates page has arrived, tables updated to reflect position, and frame marked as being in normal state.

Page Fault Handling (3)

8. Faulting instruction backed up to state it had when it began and program counter is reset
9. Faulting process is scheduled, operating system returns to routine that called it.
10. Routine reloads registers and other state information, returns to user space to continue execution

Instruction Backup

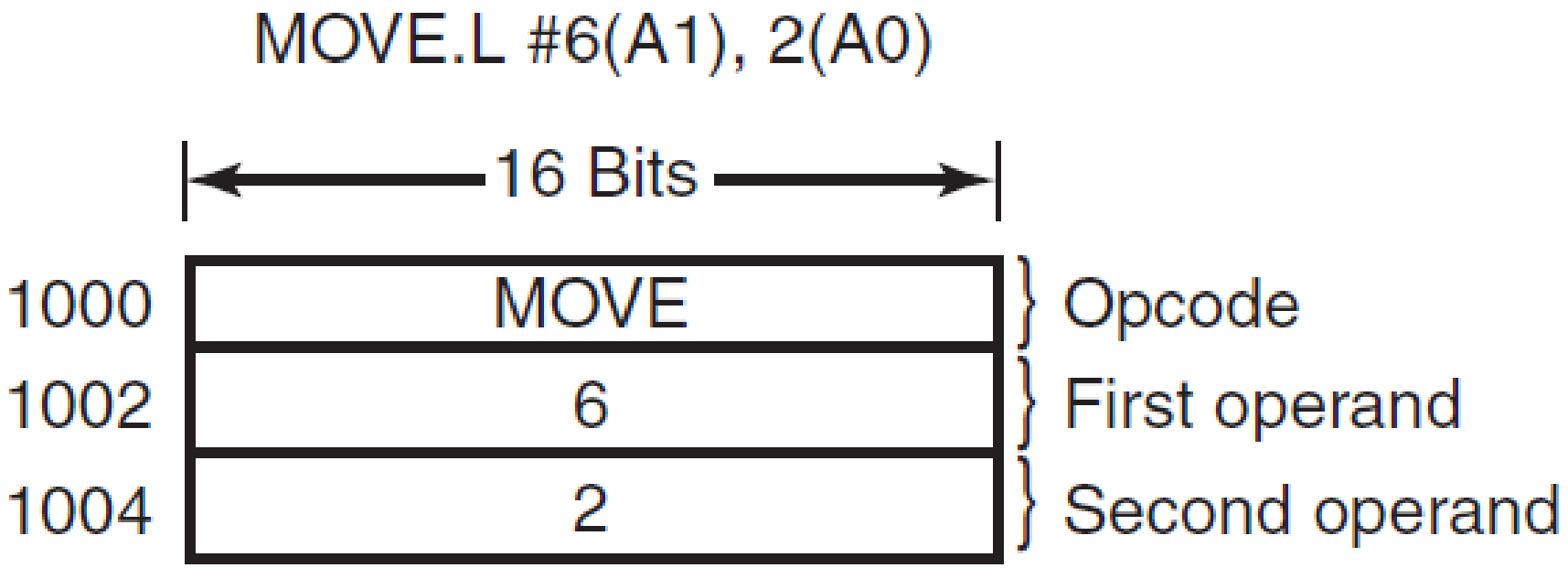


Figure 3-27. An instruction causing a page fault.

Backing Store

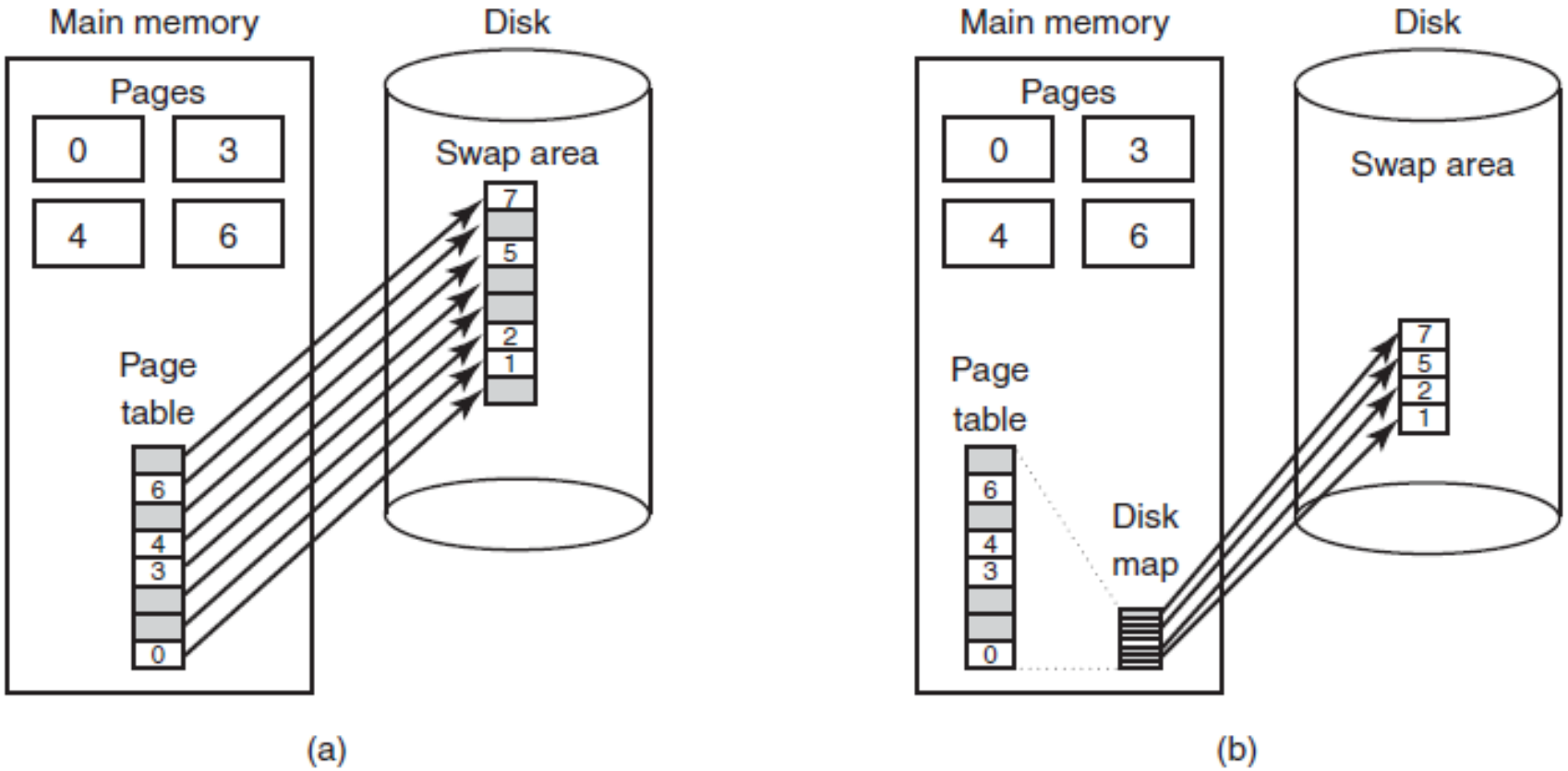


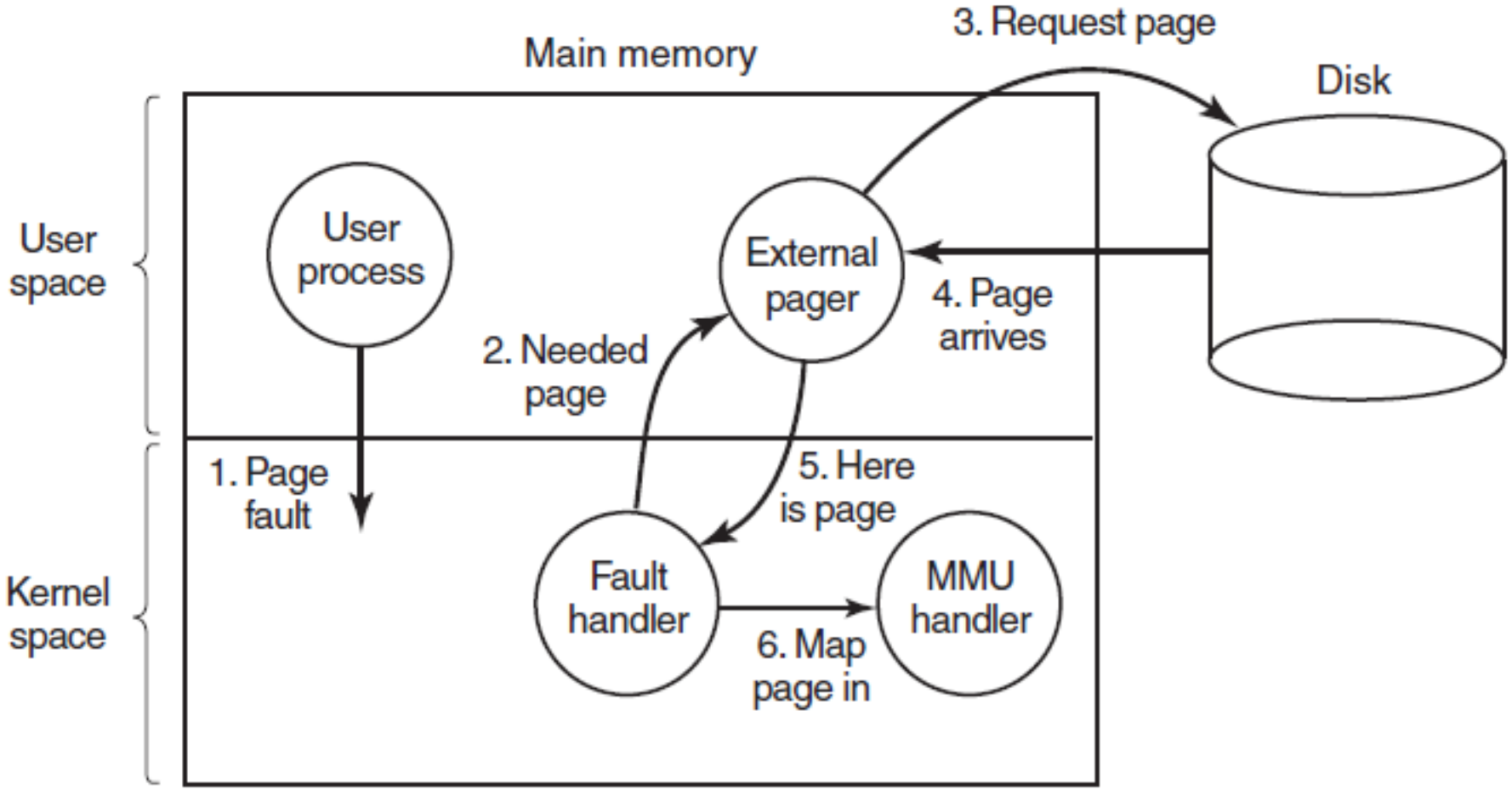
Figure 3-28. (a) Paging to a static swap area. (b) Backing up pages dynamically.

Separation of Policy and Mechanism (1)

Memory management system is divided into three parts

1. A low-level MMU handler.
2. A page fault handler that is part of the kernel.
3. An external pager running in user space.

Separation of Policy and Mechanism (2)



3-29. Page fault handling with an external pager.

Segmentation (1)

Examples of tables generated by compiler:

1. The source text being saved for the printed listing
2. The symbol table, names and attributes of variables.
3. The table containing integer and floating-point constants used.
4. The parse tree, syntactic analysis of the program.
5. The stack used for procedure calls within compiler.

Segmentation (2)

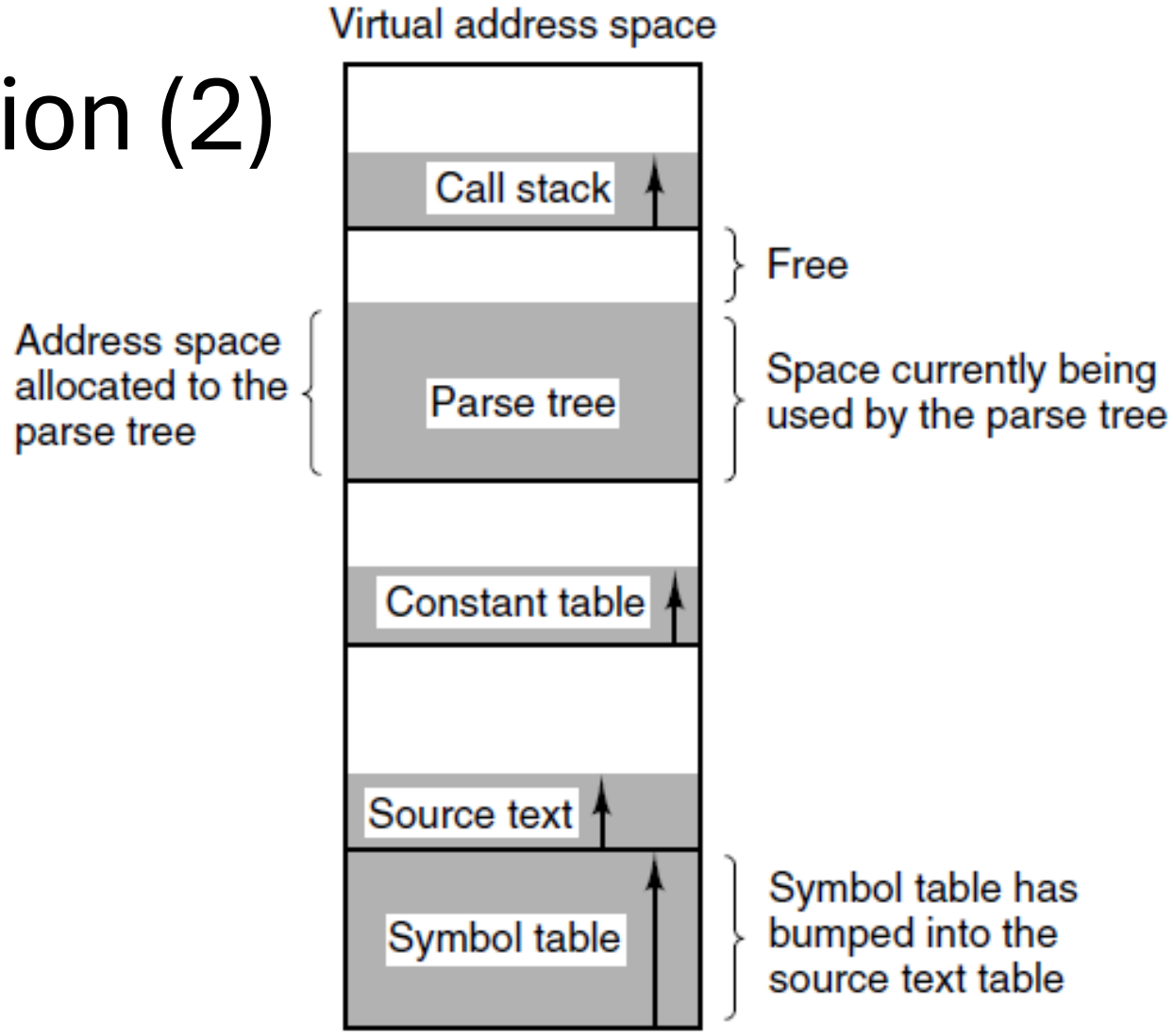


Figure 3-30. In a one-dimensional address space with growing tables, one table may bump into another.

Segmentation (3)

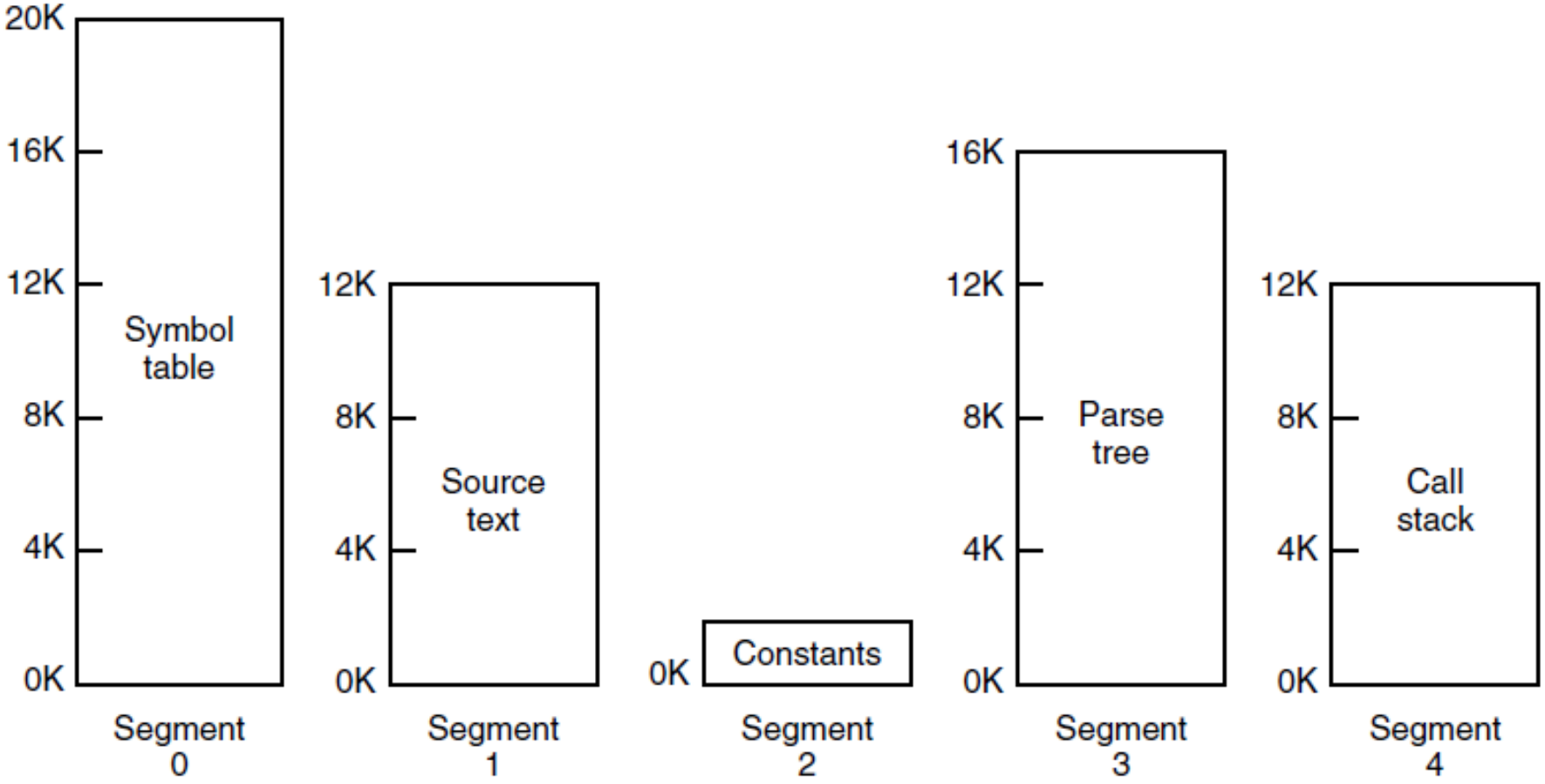


Figure 3-31. A segmented memory allows each table to grow or shrink independently of the other tables.

Segmentation (4)

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

Figure 3-32. Comparison of paging and segmentation

Implementation of Pure Segmentation

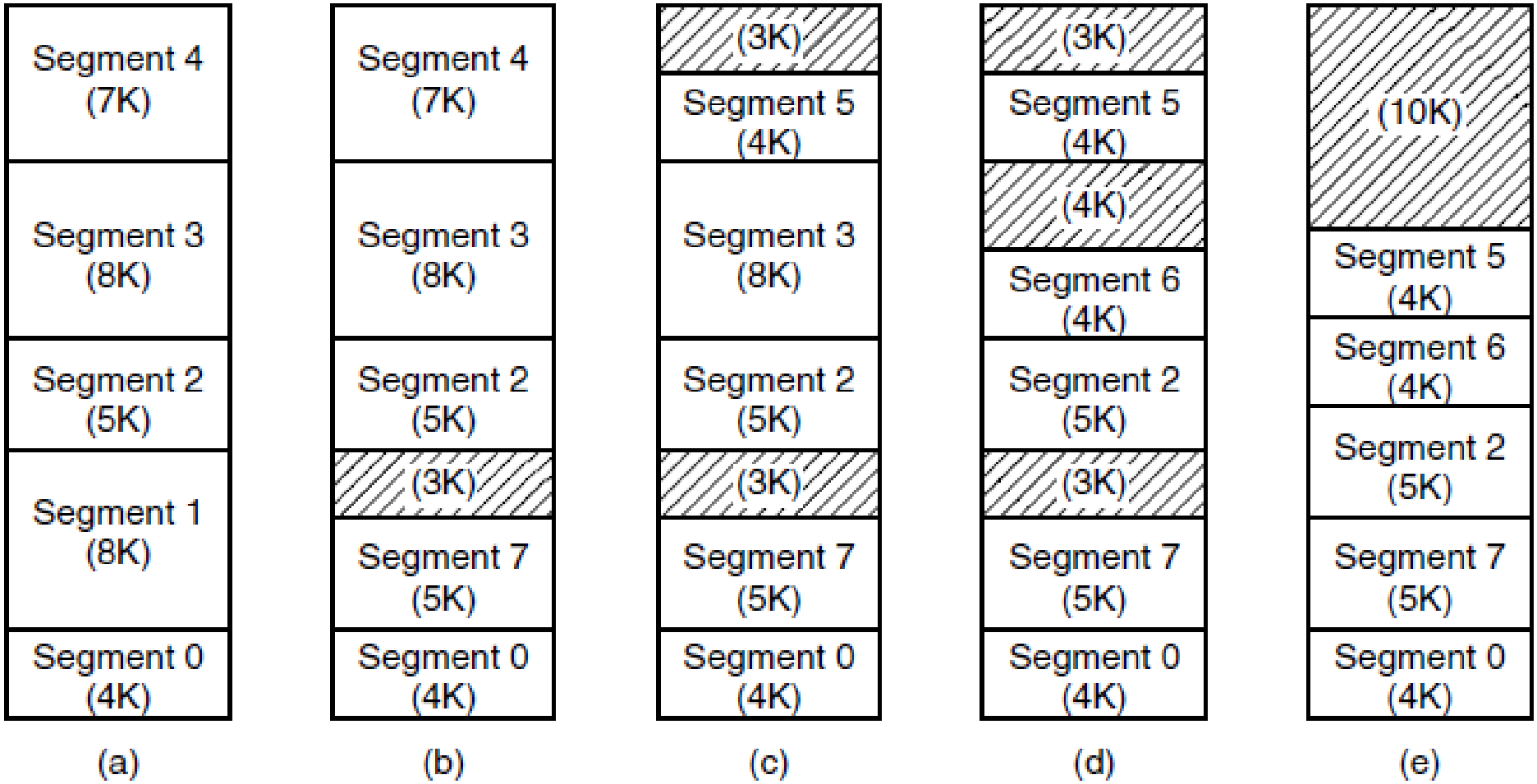


Figure 3-33. (a)-(d) Development of checkerboarding. (e) Removal of the checkerboarding by compaction.

Segmentation with Paging: MULTICS (1)

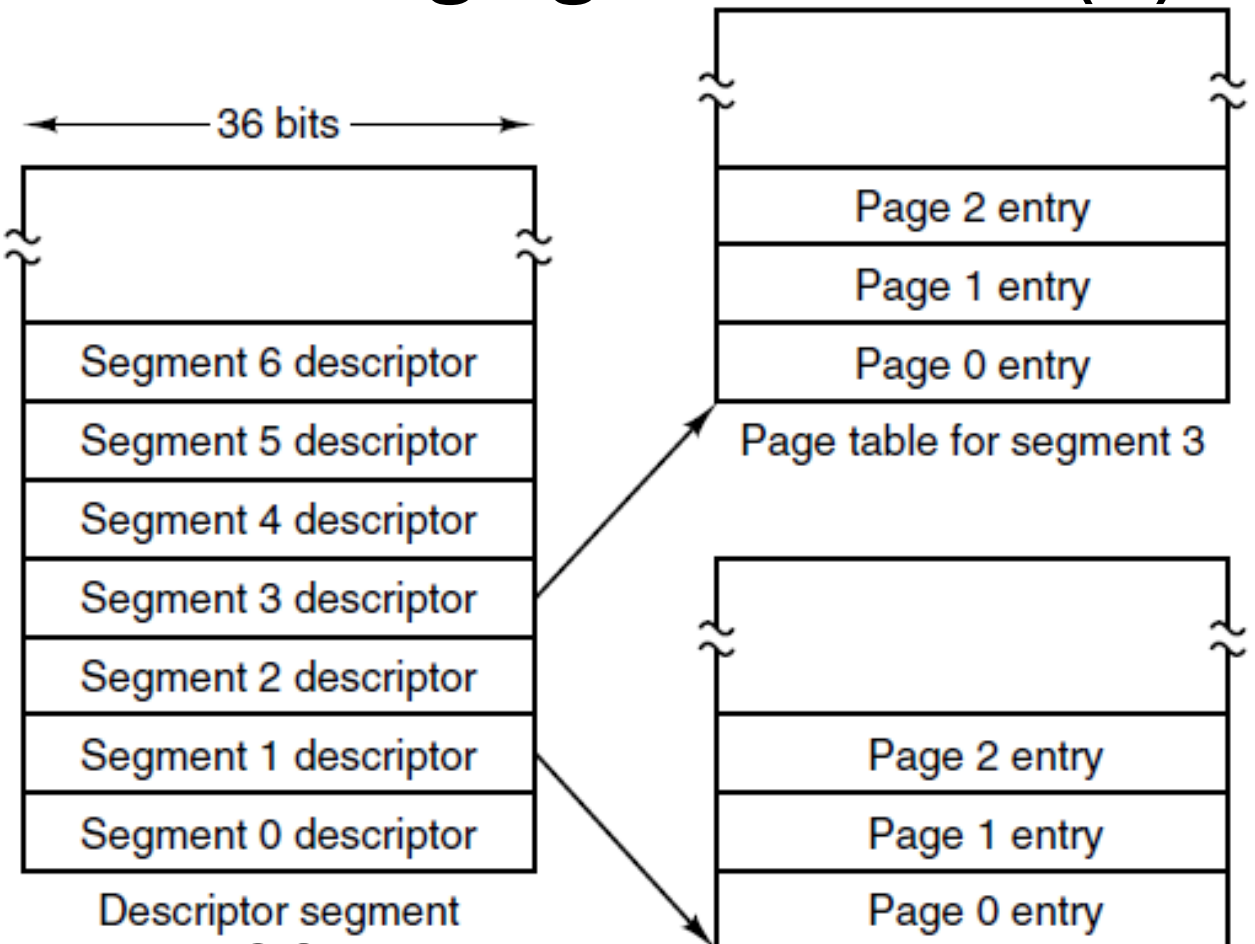


Figure 3-34. The MULTICS virtual memory. (a) The descriptor segment pointed to the page tables.

Segmentation with Paging: MULTICS (2)

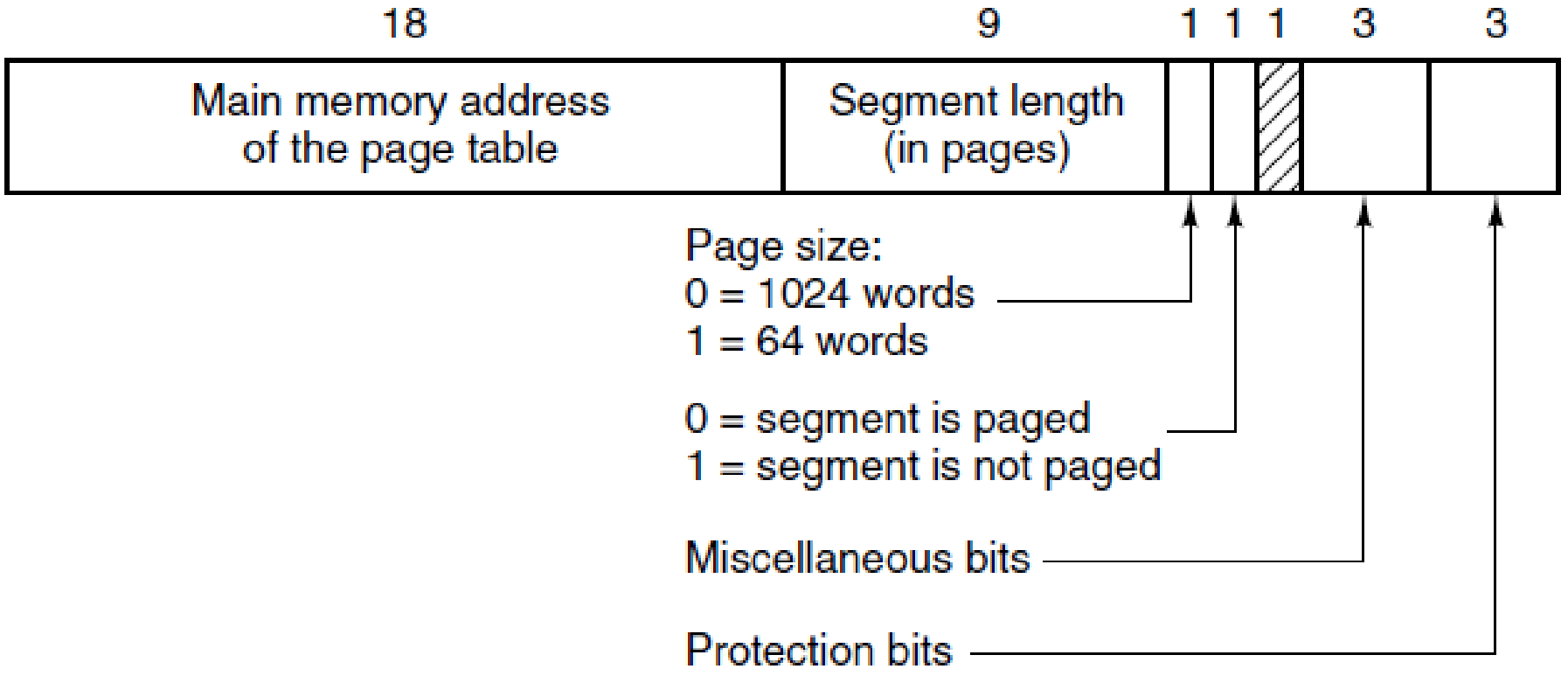


Figure 3-34. The MULTICS virtual memory. (b) A segment descriptor. The numbers are the field lengths.

Segmentation with Paging: MULTICS (3)

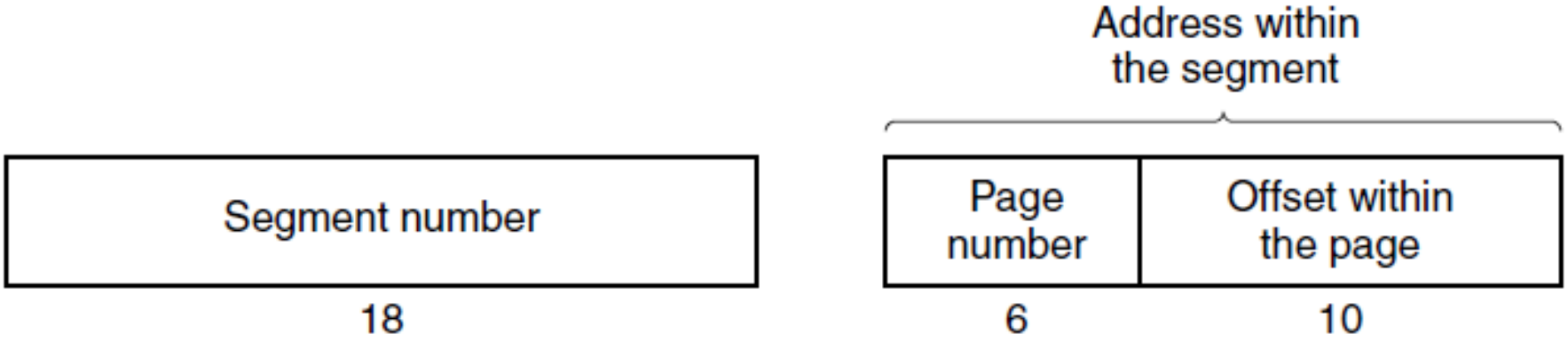


Figure 3-35. A 34-bit MULTICS virtual address.

Segmentation with Paging: MULTICS (4)

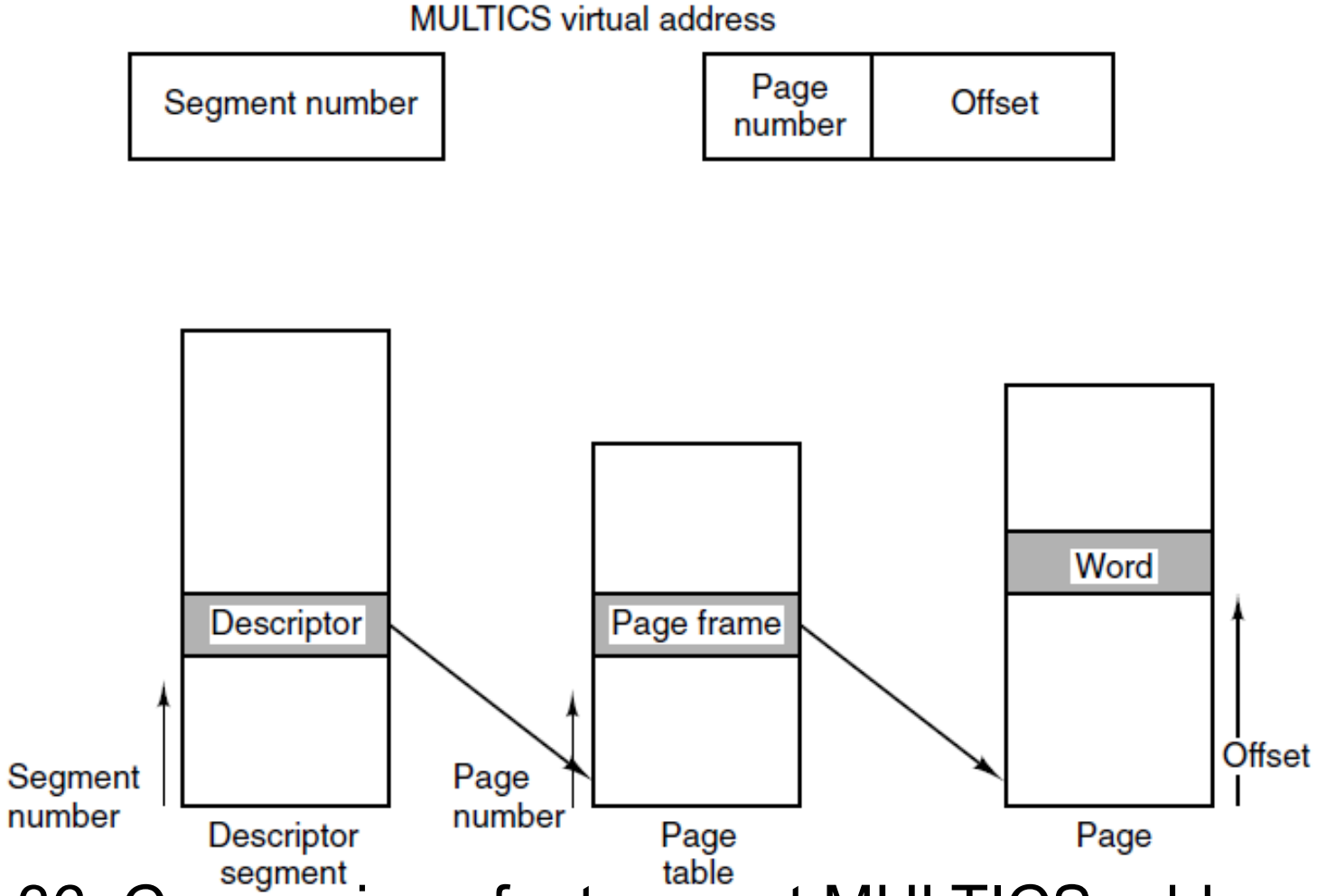


Figure 3-36. Conversion of a two-part MULTICS address into a main memory address.

Segmentation with Paging: MULTICS (5)

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

Figure 3-37. A simplified version of the MULTICS TLB. The existence of two page sizes made the actual TLB more complicated.

Segmentation with Paging: The Intel x86 (1)

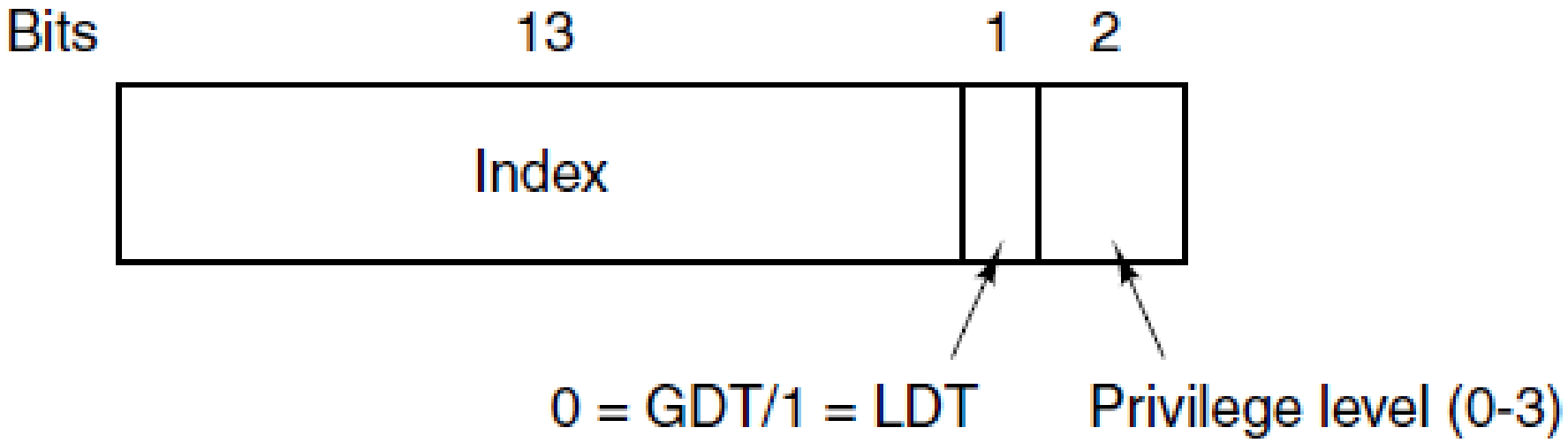


Figure 3-38. An x86 selector.

Segmentation with Paging: The Intel x86 (2)

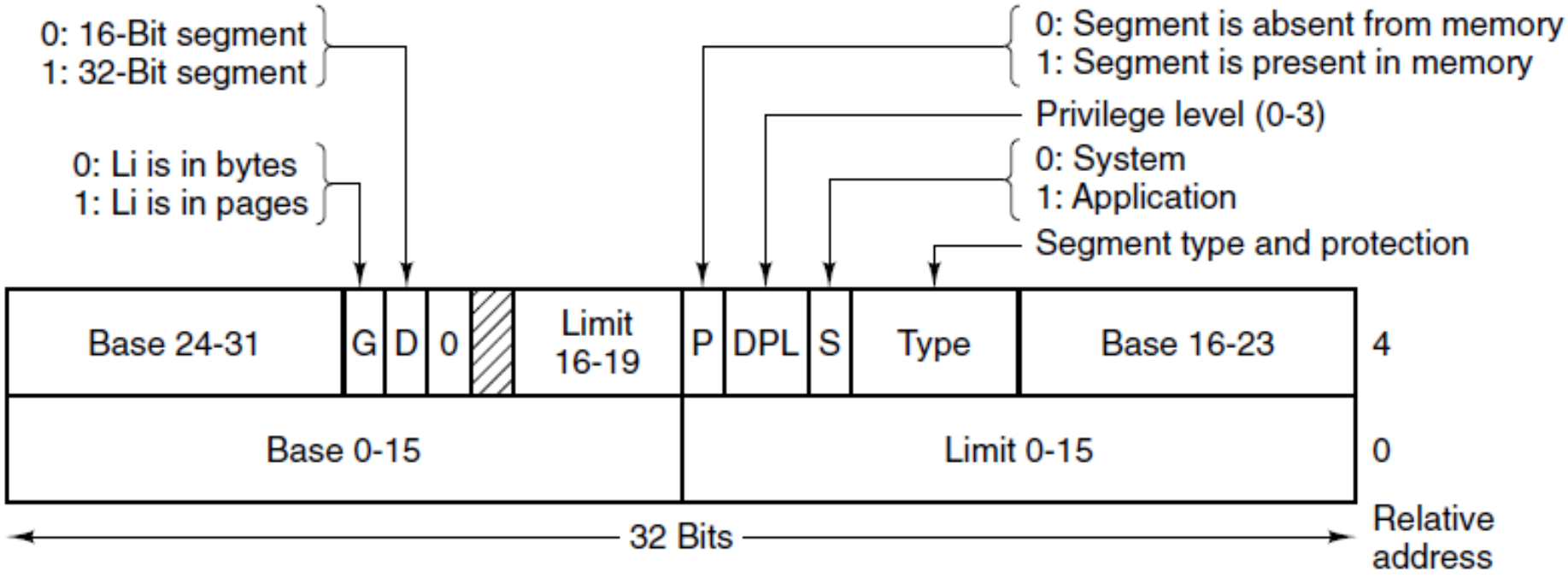


Figure 3-39. x86 code segment descriptor. Data segments differ slightly.

Segmentation with Paging: The Intel x86 (3)

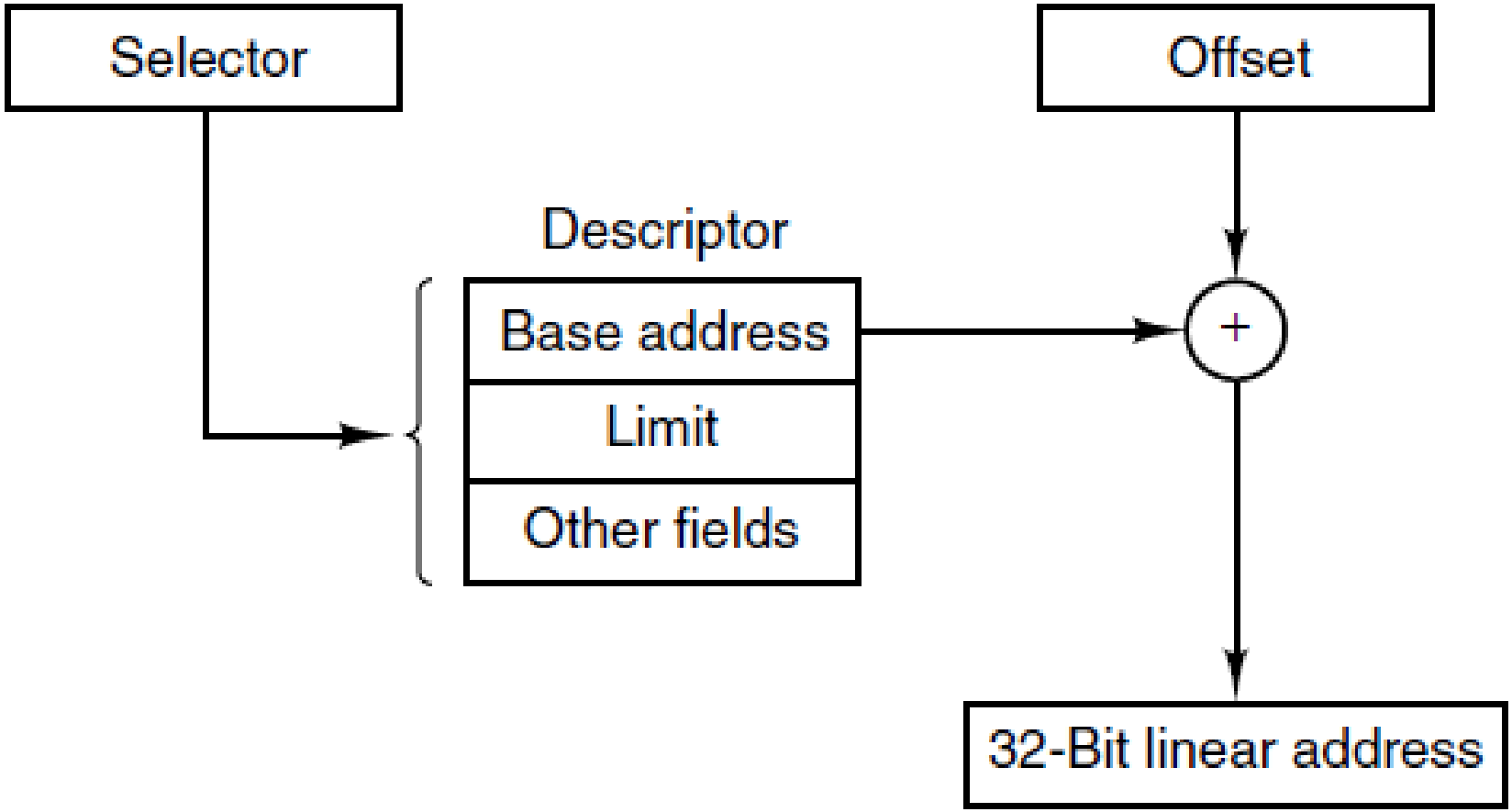


Figure 3-40. Conversion of a (selector, offset) pair to a linear address.

Segmentation with Paging: The Intel x86 (4)

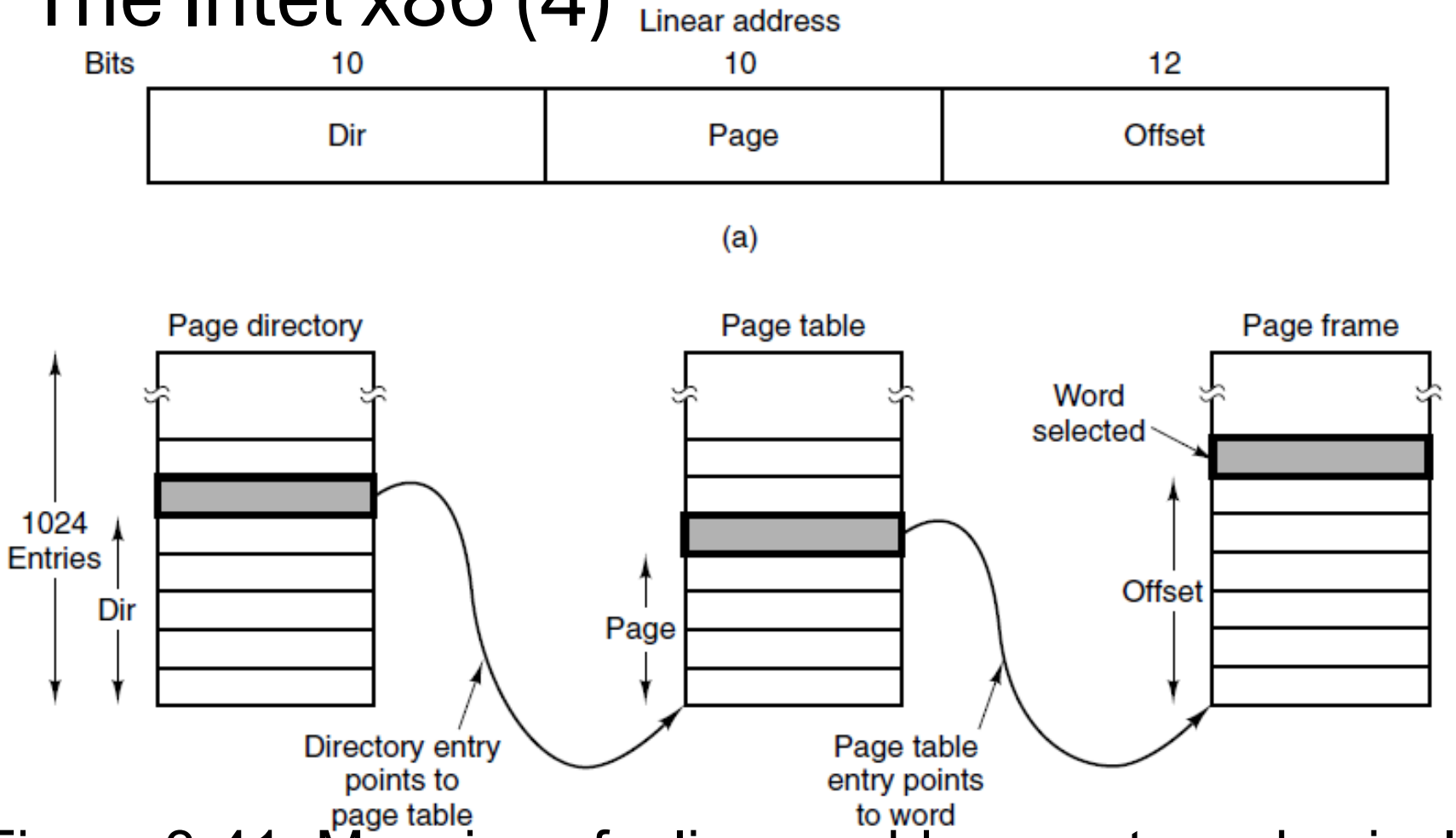


Figure 3-41. Mapping of a linear address onto a physical address.