4

MEMORY MANAGEMENT

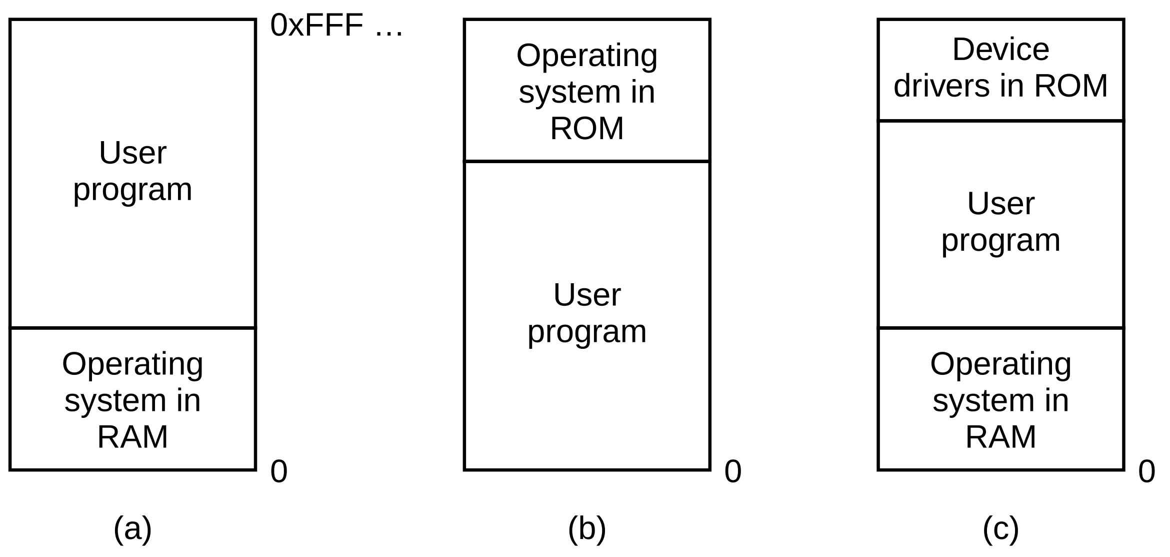


Fig. 4-1. Three simple ways of organizing memory with an operating system and one user process. Other possibilities also exist.

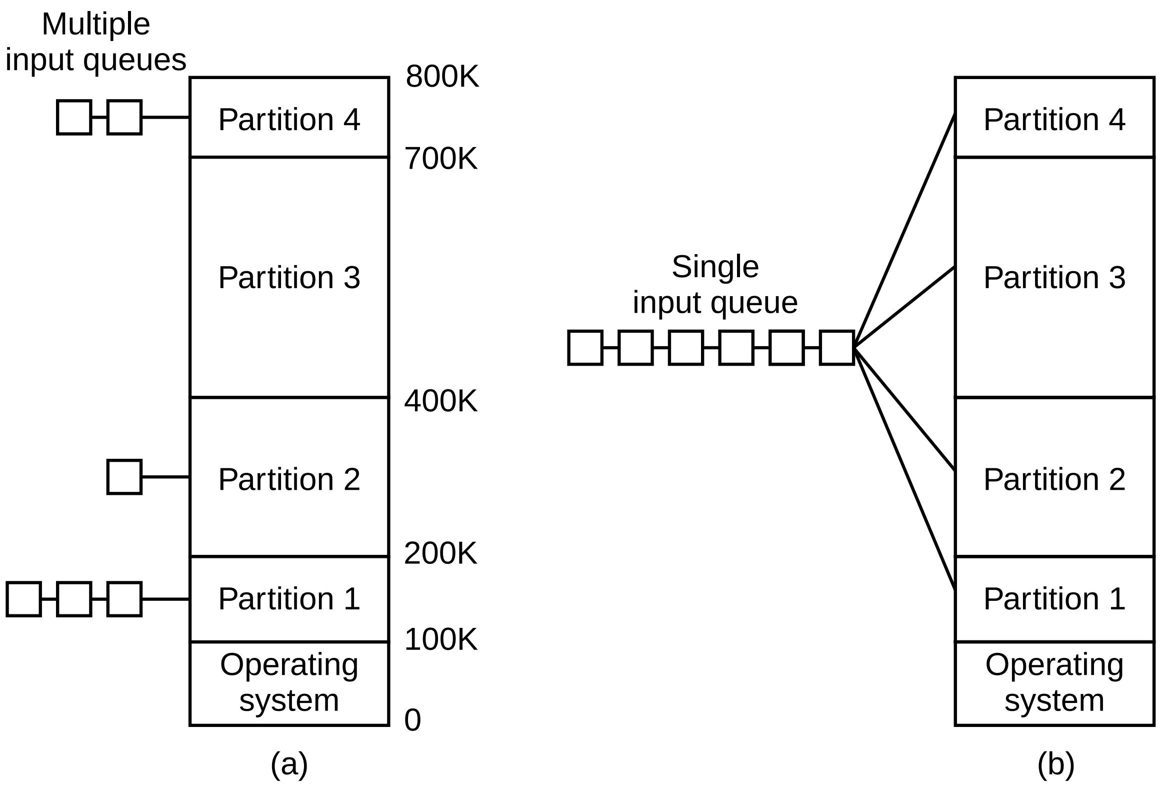


Fig. 4-2. (a) Fixed memory partitions with separate input queues for each partition. (b) Fixed memory partitions with a single input queue.

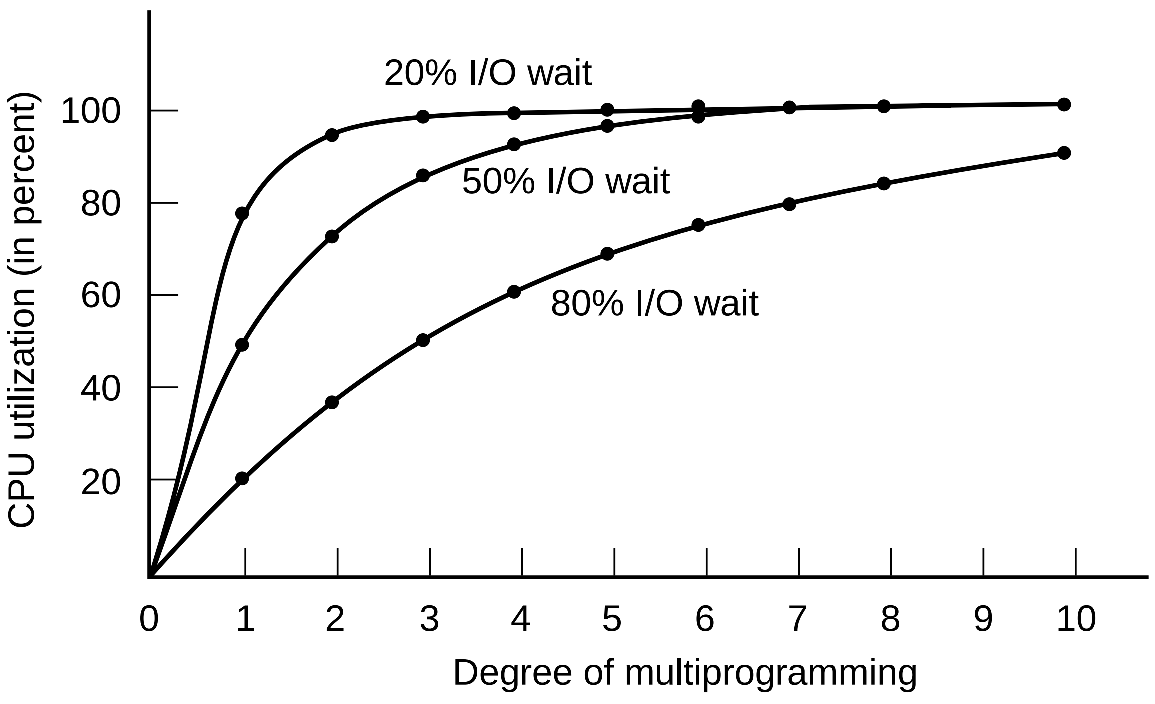


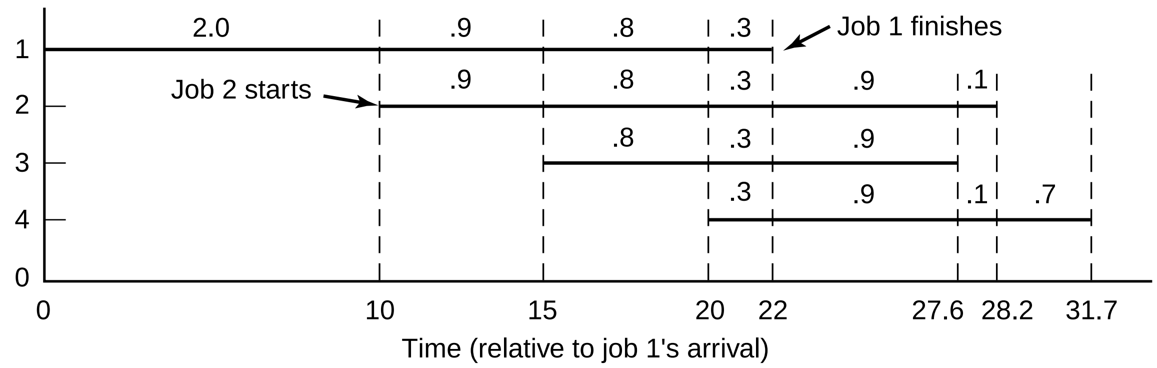
Fig. 4-3. CPU utilization as a function of the number of processes in memory.

Job	Arrival time	CPU minutes needed
1	10:00	4
2	10:10	3
3	10:15	2
4	10:20	2

(a)

	# Processes			
	1	2	3	4
CPU idle	.80	.64	.51	.41
CPU busy	.20	.36	.49	.59
CPU/process	.20	.18	.16	.15

(b)



(c)

Fig. 4-4. (a) Arrival and work requirements of four jobs. (b) CPU utilization for 1 to 4 jobs with 80 percent I/O wait. (c) Sequence of events as jobs arrive and finish. The numbers above the horizontal lines show how much CPU time, in minutes, each job gets in each interval.

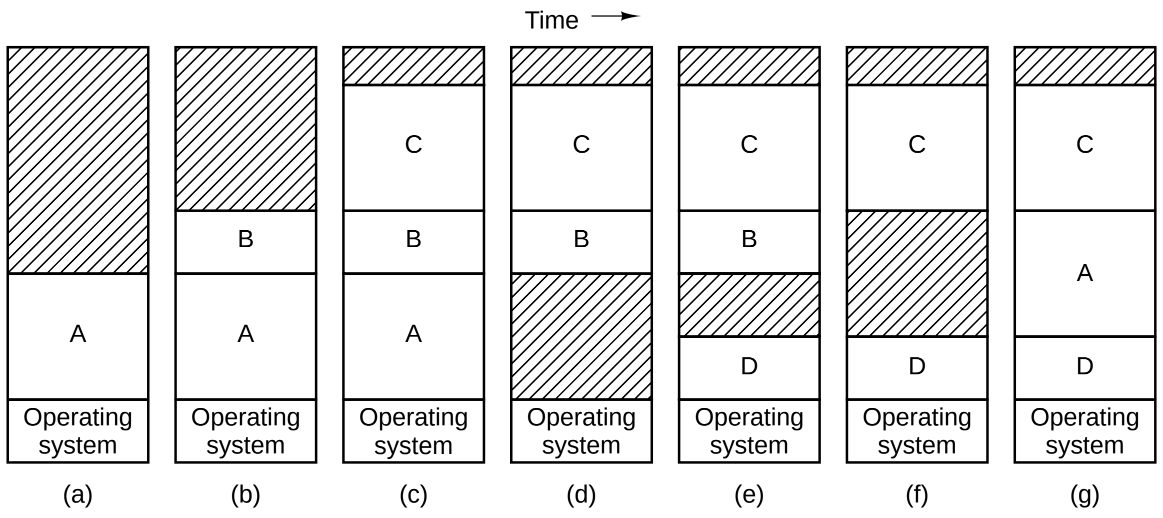


Fig. 4-5. Memory allocation changes as processes come into memory and leave it. The shaded regions are unused memory.

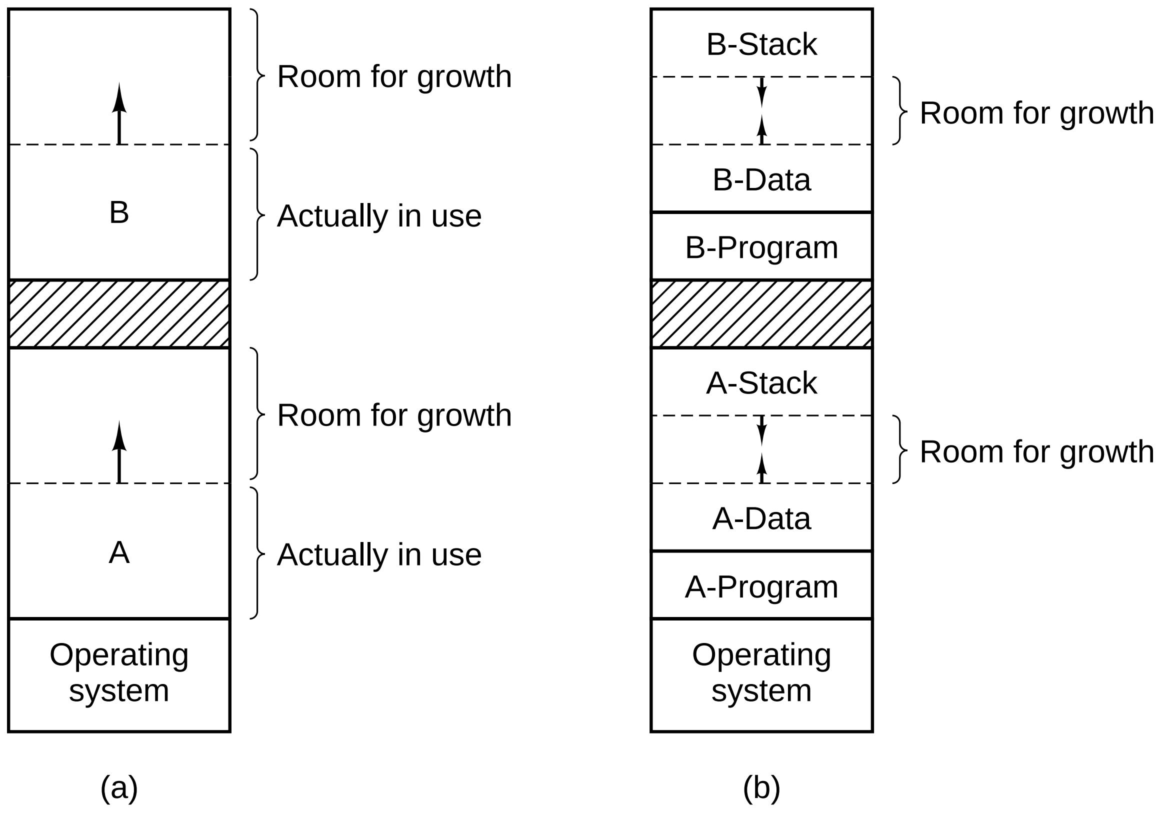


Fig. 4-6. (a) Allocating space for a growing data segment. (b) Allocating space for a growing stack and a growing data segment.

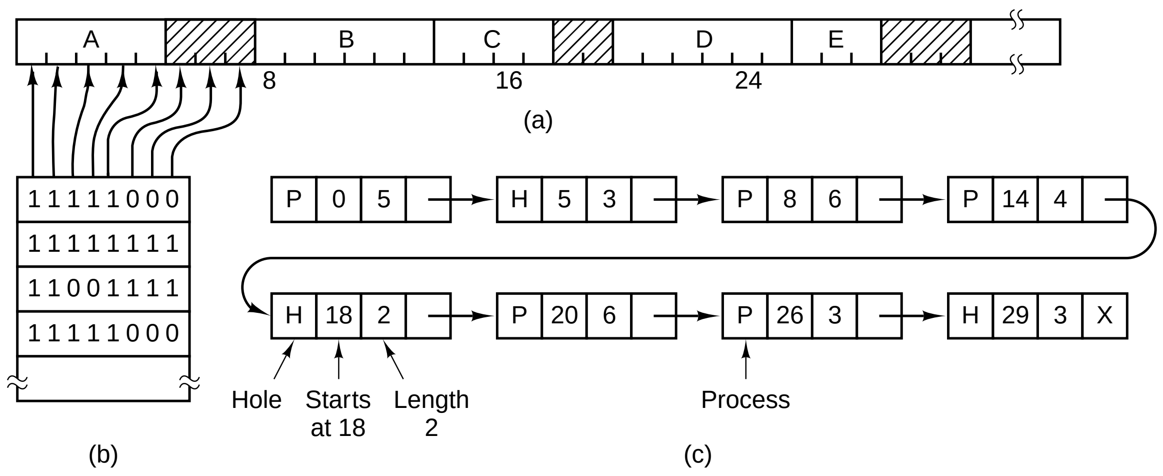


Fig. 4-7. (a) A part of memory with five processes and three holes. The tick marks show the memory allocation units. The shaded regions (0 in the bitmap) are free. (b) The corresponding bitmap. (c) The same information as a list.

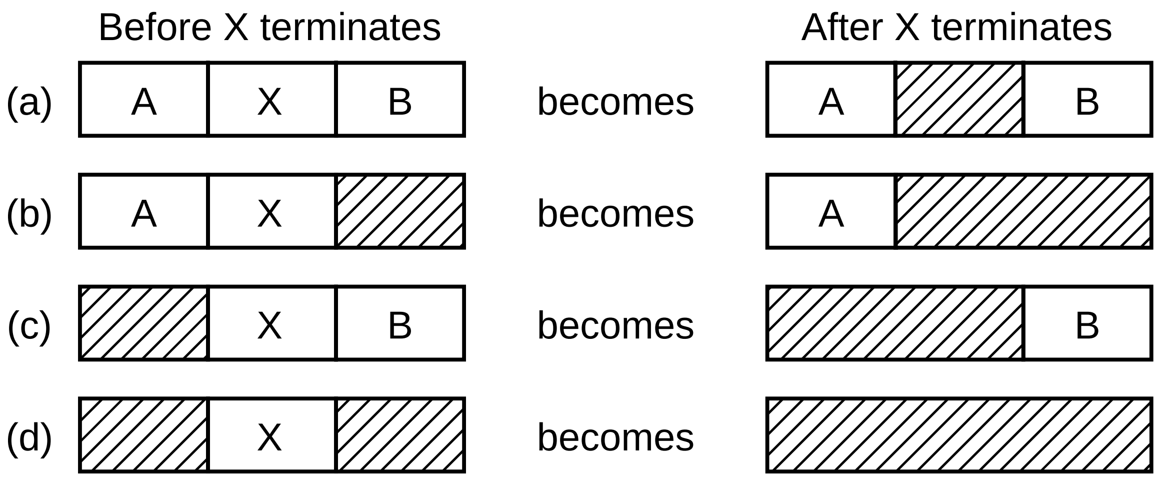


Fig. 4-8. Four neighbor combinations for the terminating process, *X*.

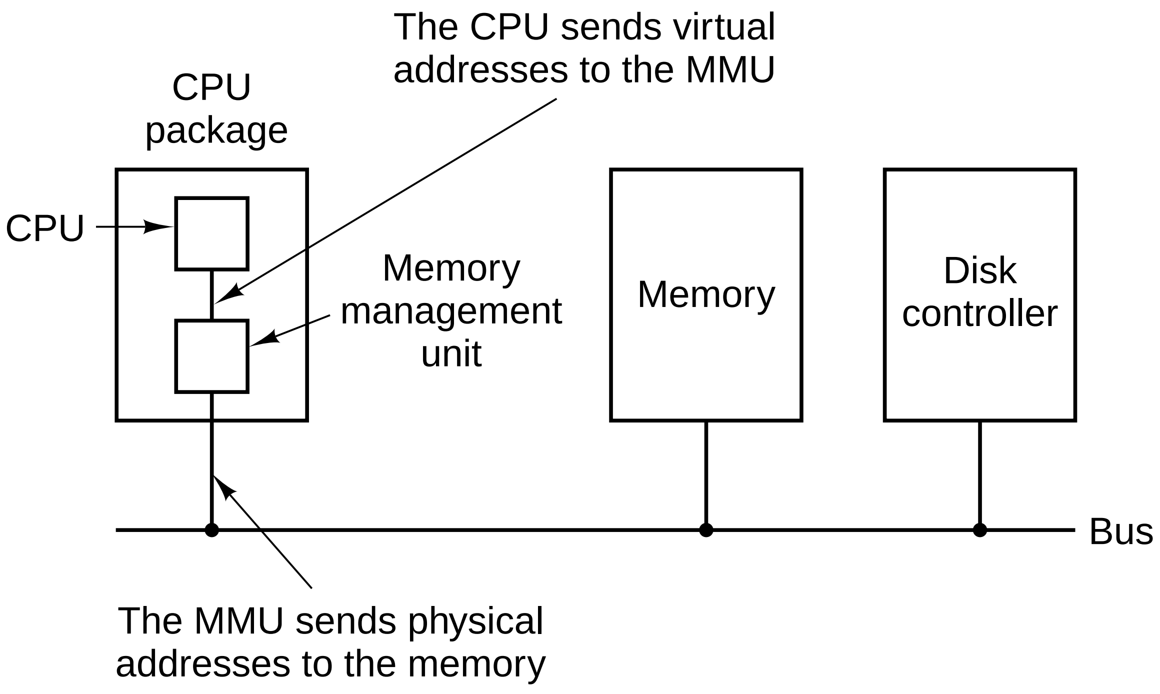

The CPU sends virtual addresses to the MMU
CPU package
CPU
Memory management unit
Memory
Disk controller
Bus
The MMU sends physical addresses to the memory

Fig. 4-9. 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 in years gone by.

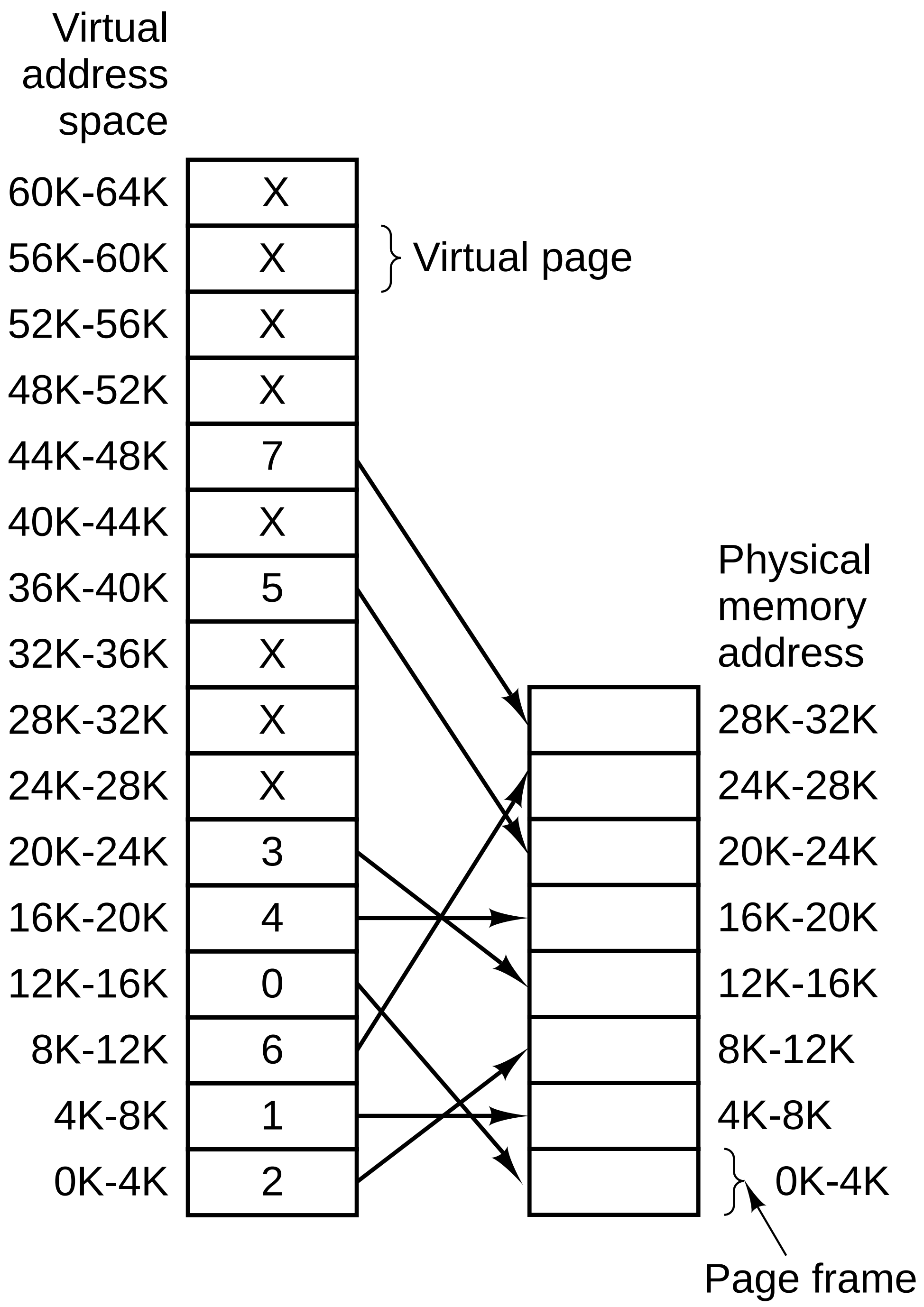


Fig. 4-10. The relation between virtual addresses and physical memory addresses is given by the page table.

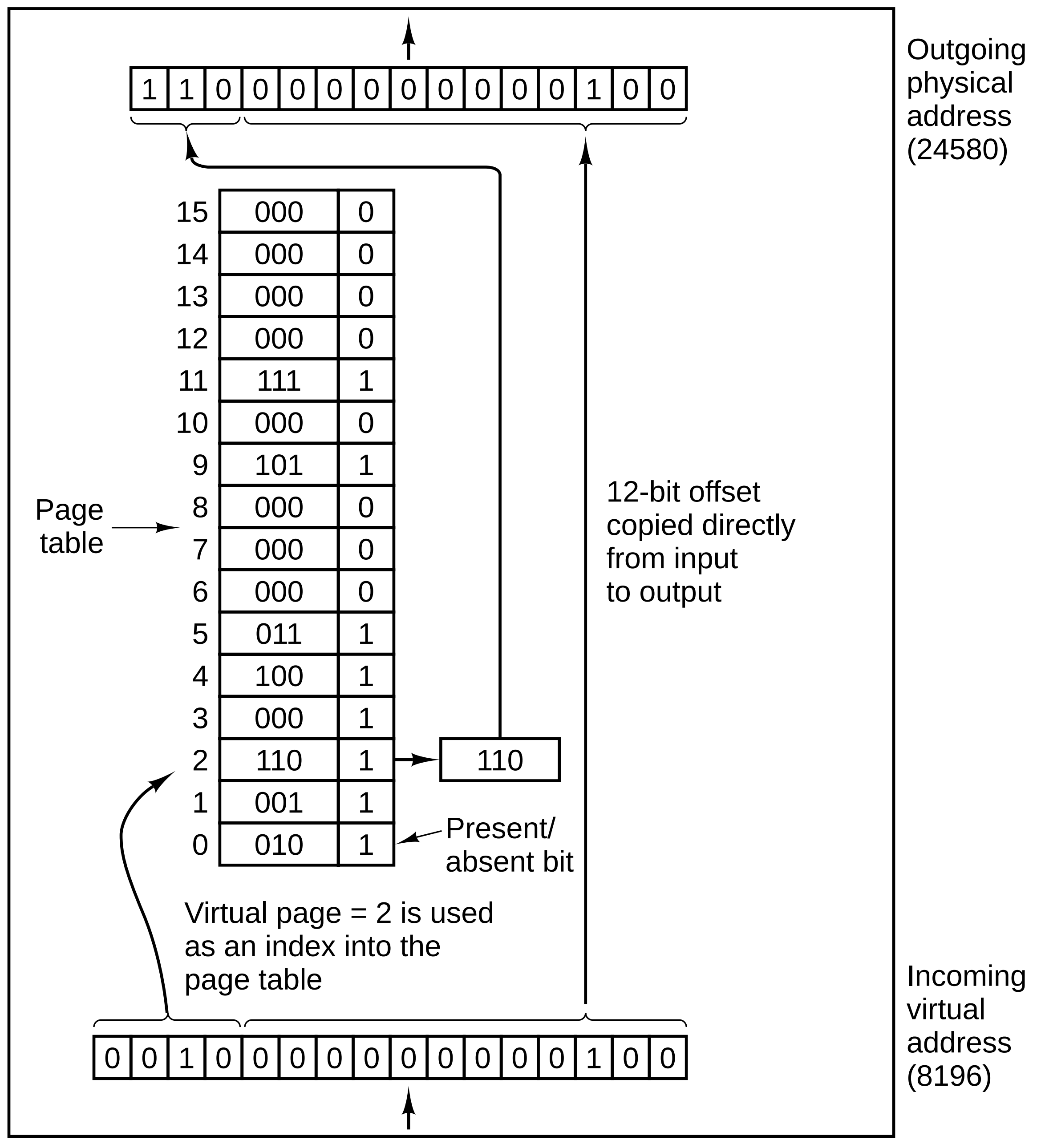


Fig. 4-11. The internal operation of the MMU with 16 4-KB pages.

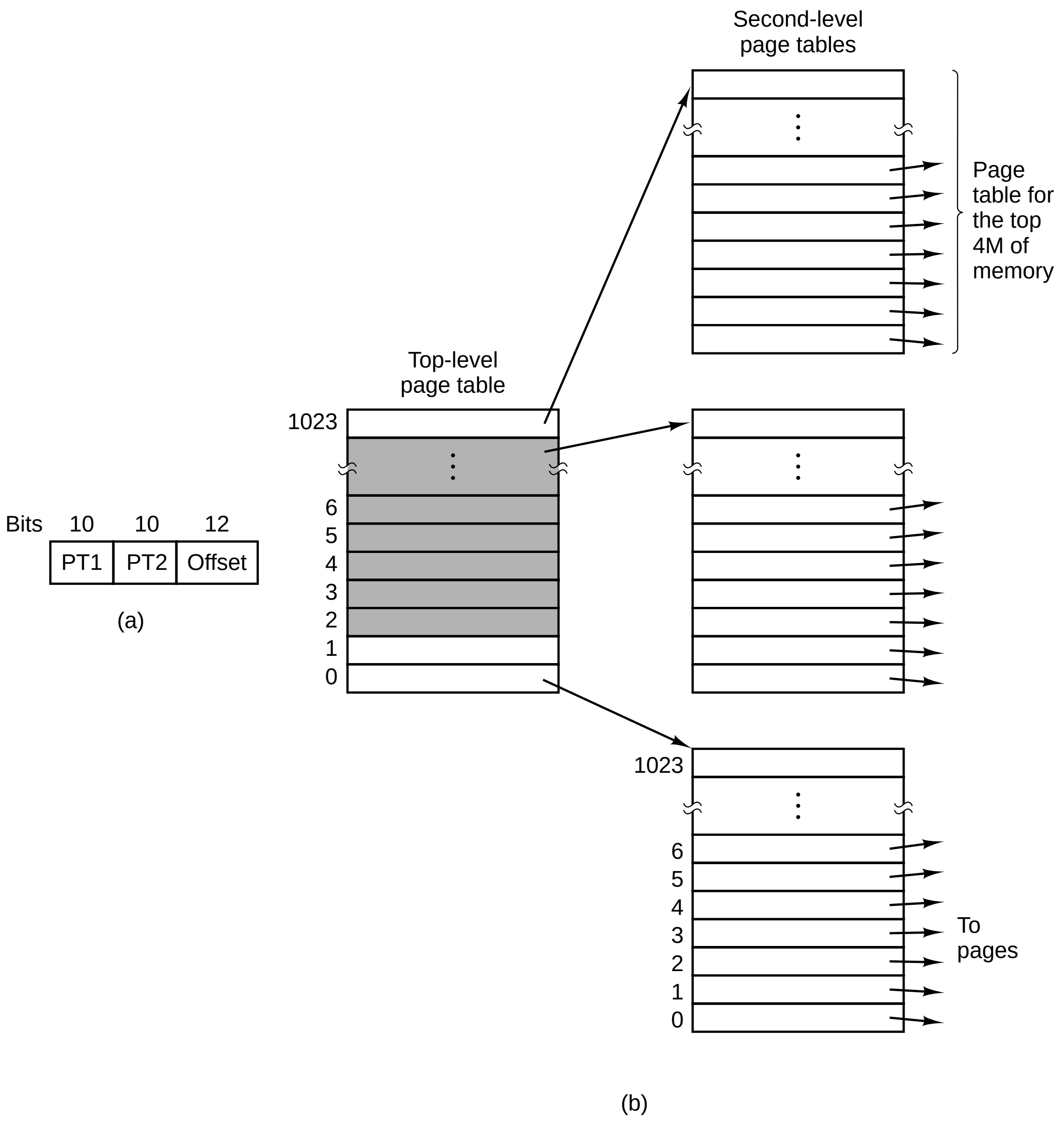


Fig. 4-12. (a) A 32-bit address with two page table fields.
(b) Two-level page tables.

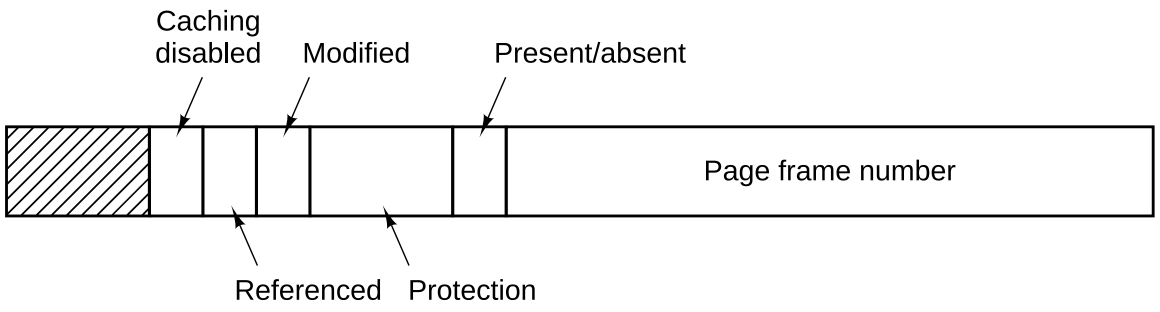


Fig. 4-13. A typical page table entry.

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

Fig. 4-14. A TLB to speed up paging.

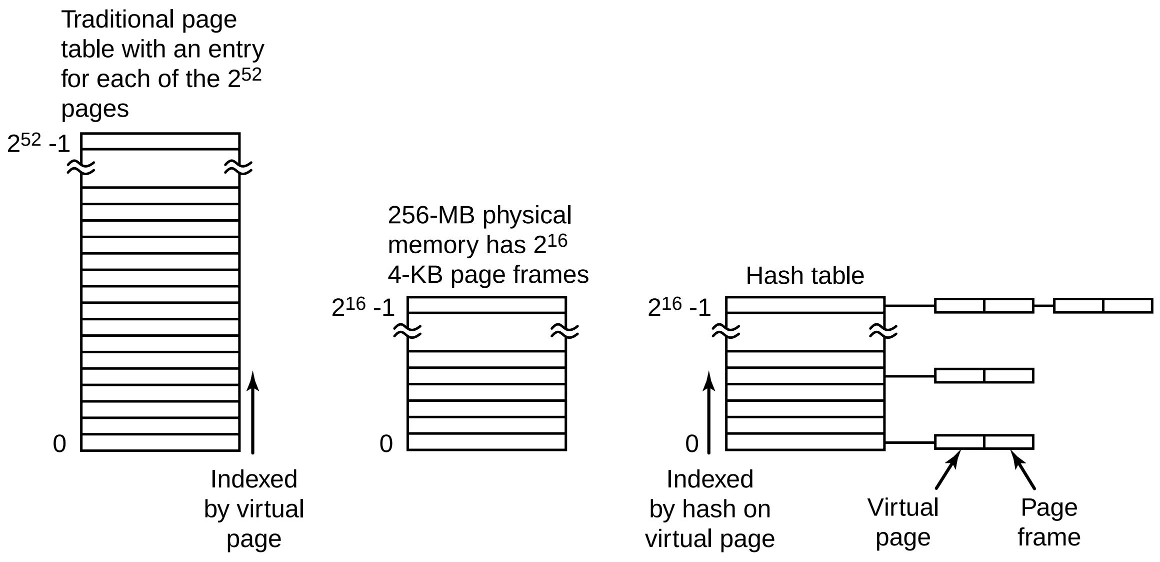

Traditional page table with an entry for each of the 2^{52} pages
2^{52} -1
0
Indexed by virtual page
256-MB physical memory has 2^{16} 4-KB page frames
2^{16} -1
0
Hash table
2^{16} -1
0
Indexed by hash on virtual page
Virtual page
Page frame

Fig. 4-15. Comparison of a traditional page table with an inverted page table.

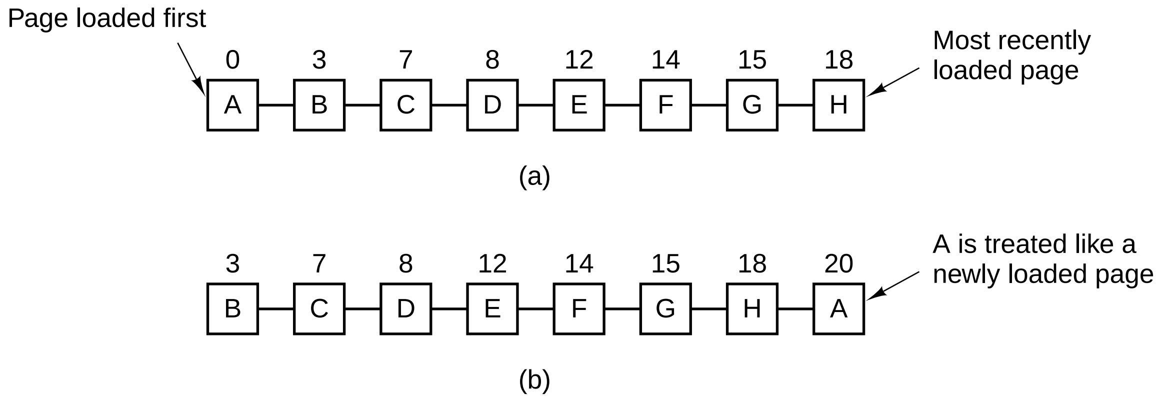
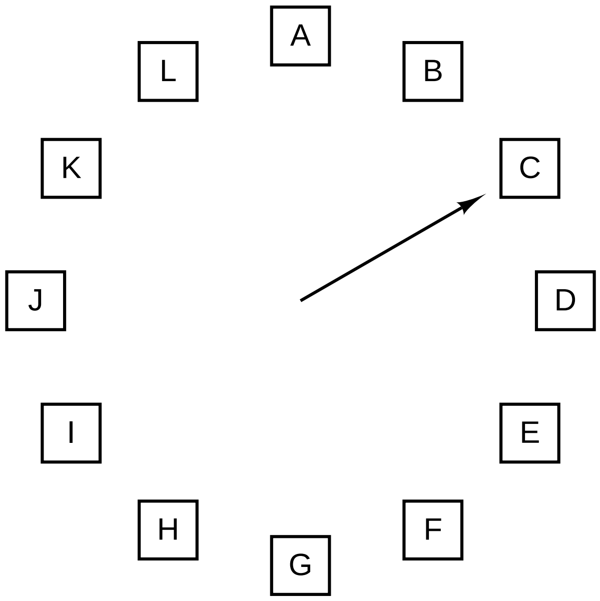


Fig. 4-16. Operation of second chance. (a) Pages sorted in FIFO order. (b) Page list if a page fault occurs at time 20 and *A* has its *R* bit set. The numbers above the pages are their loading times.



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

Fig. 4-17. The clock page replacement algorithm.

$$\begin{array}{c|cccc|} & \multicolumn{4}{c}{\text{Page}} \\ & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 \\ \hline \end{array}$$

(a)

$$\begin{array}{|cccc|} \multicolumn{4}{c}{\text{Page}} \\ 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \hline \end{array}$$

(b)

$$\begin{array}{|cccc|} \multicolumn{4}{c}{\text{Page}} \\ 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ \hline \end{array}$$

(c)

$$\begin{array}{|cccc|} \multicolumn{4}{c}{\text{Page}} \\ 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ \hline \end{array}$$

(d)

$$\begin{array}{|cccc|} \multicolumn{4}{c}{\text{Page}} \\ 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ \hline \end{array}$$

(e)

$$\begin{array}{|cccc|} \hline 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ \hline \end{array}$$

(f)

$$\begin{array}{|cccc|} \hline 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ \hline \end{array}$$

(g)

$$\begin{array}{|cccc|} \hline 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ \hline \end{array}$$

(h)

$$\begin{array}{|cccc|} \hline 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ \hline \end{array}$$

(i)

$$\begin{array}{|cccc|} \hline 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ \hline \end{array}$$

(j)

Fig. 4-18. LRU using a matrix when pages are referenced in the order 0, 1, 2, 3, 2, 1, 0, 3, 2, 3.

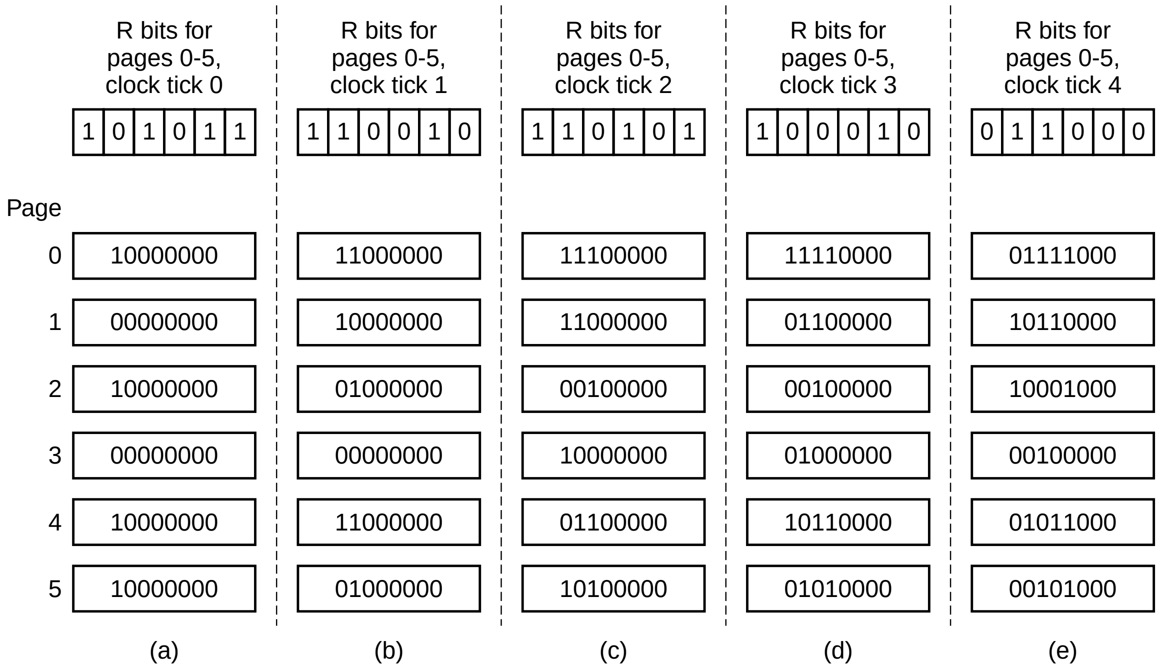


Fig. 4-19. The aging algorithm simulates LRU in software. Shown are six pages for five clock ticks. The five clock ticks are represented by (a) to (e).

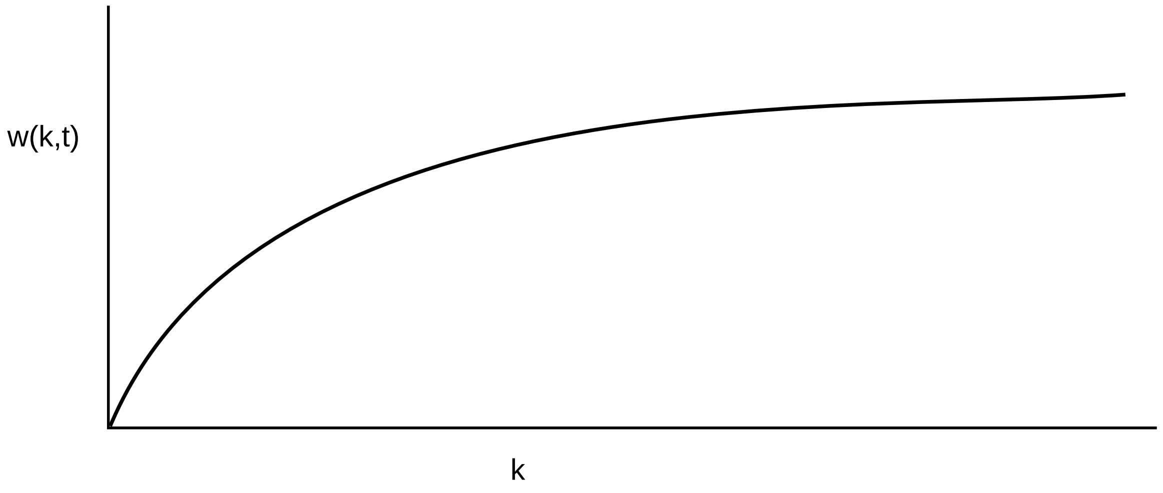


Fig. 4-20. The working set is the set of pages used by the *k* most recent memory references. The function *w*(*k*, *t*) is the size of the working set at time *t*.

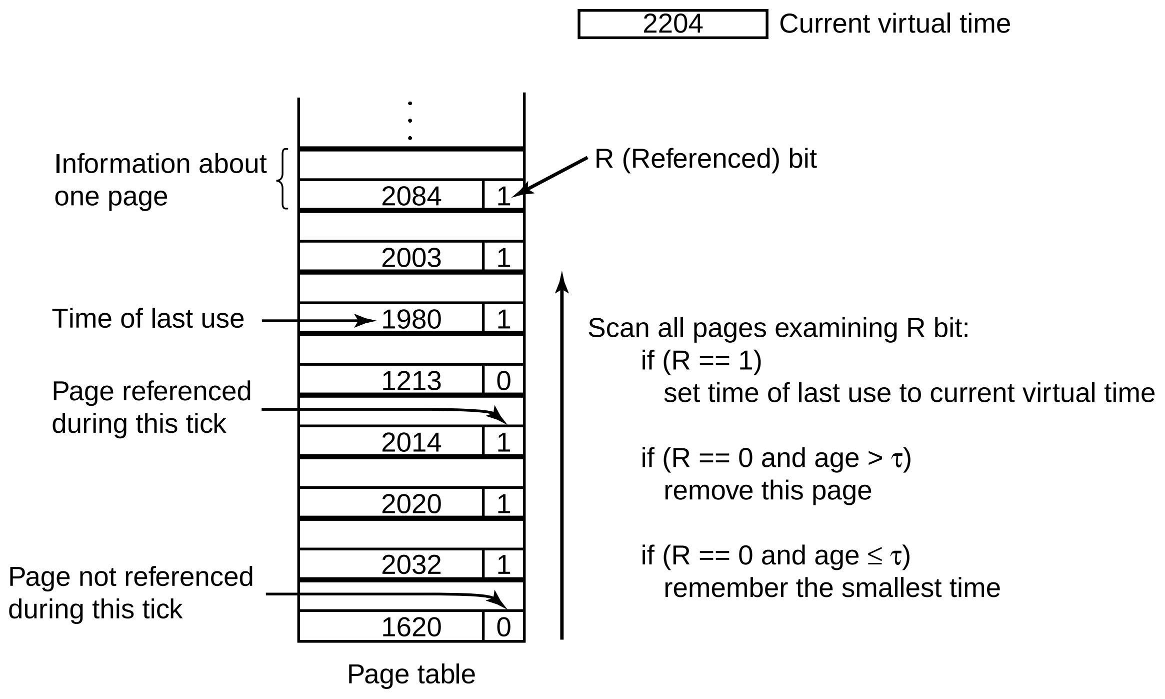

2204
Current virtual time
Information about one page
R (Referenced) bit
2084 1
2003 1
Time of last use
1980 1
1213 0
Page referenced during this tick
2014 1
2020 1
2032 1
Page not referenced during this tick
1620 0
Page table
Scan all pages examining R bit:
if (R == 1)
set time of last use to current virtual time
if (R == 0 and age > τ)
remove this page
if (R == 0 and age ≤ τ)
remember the smallest time

Fig. 4-21. The working set algorithm.

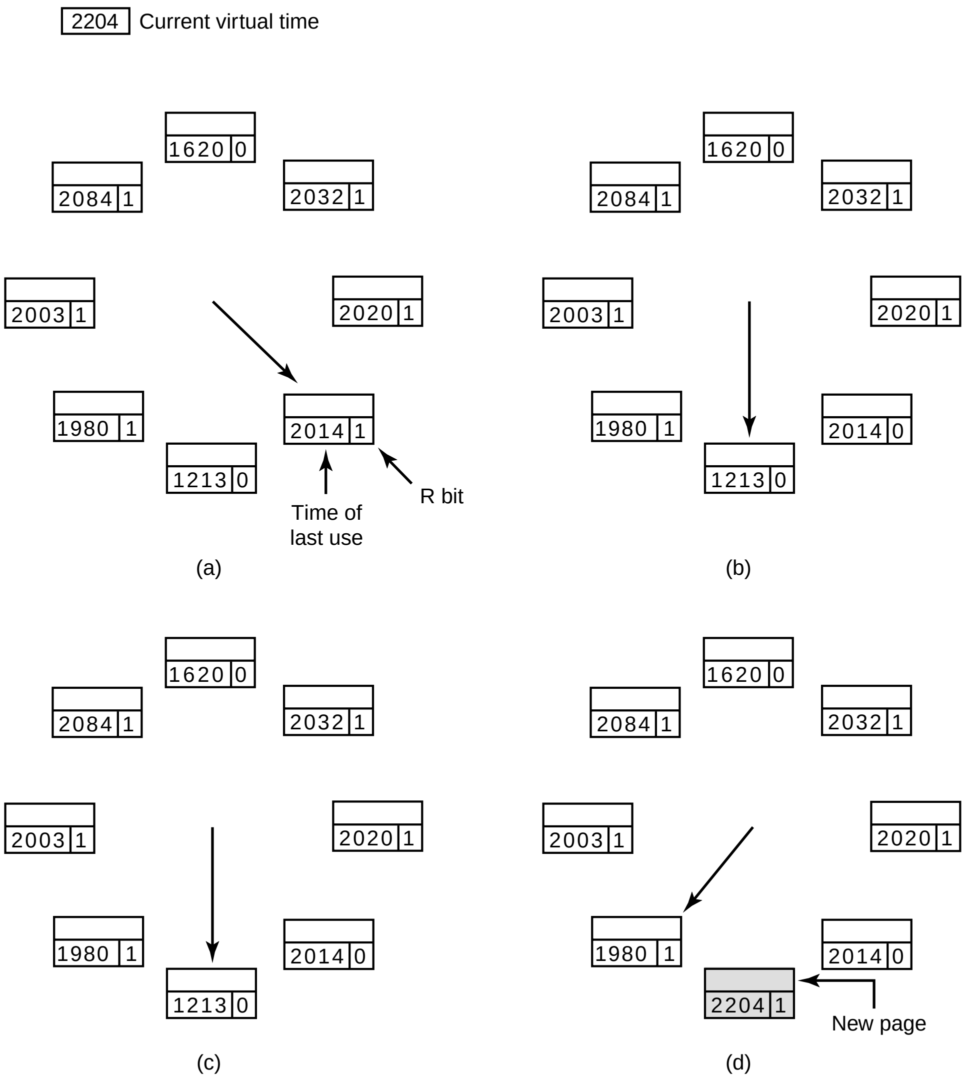


Fig. 4-22. Operation of the WSClock algorithm. (a) and (b) give an example of what happens when $R = 1$. (c) and (d) give an example of $R = 0$.

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

Fig. 4-23. Page replacement algorithms discussed in the text.

All pages frames initially empty

		0	1	2	3	0	1	4	0	1	2	3	4	
Youngest page		0	1	2	3	0	1	4	4	4	2	3	3	
			0	1	2	3	0	1	1	1	4	2	2	
Oldest page				0	1	2	3	0	0	0	1	4	4	
		P	P	P	P	P	P	P			P	P		9 Page faults

(a)

		0	1	2	3	0	1	4	0	1	2	3	4	
Youngest page		0	1	2	3	3	3	4	0	1	2	3	4	
			0	1	2	2	2	3	4	0	1	2	3	
Oldest page				0	1	1	1	2	3	4	0	1	2	
					0	0	0	1	2	3	4	0	1	
		P	P	P	P			P	P	P	P	P	P	10 Page faults

(b)

Fig. 4-24. Belady's anomaly. (a) FIFO with three page frames. (b) FIFO with four page frames. The *P*'s show which page references cause page faults.

Reference string	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	1	3	4	1
	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	1	3	4	1
		0	2	1	3	5	4	6	3	7	4	7	7	3	3	5	3	3	3	1	7	1	3	4
			0	2	1	3	5	4	6	3	3	4	4	7	7	7	5	5	5	3	3	7	1	3
				0	2	1	3	5	4	6	6	6	6	4	4	4	7	7	7	5	5	5	7	7
					0	2	1	1	5	5	5	5	5	6	6	6	4	4	4	4	4	4	5	5
						0	2	2	1	1	1	1	1	1	1	1	6	6	6	6	6	6	6	6
							0	0	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
									0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Page faults	P	P	P	P	P	P	P		P					P			P						P	
Distance string	∞	∞	∞	∞	∞	∞	∞	4	∞	4	2	3	1	5	1	2	6	1	1	4	2	3	5	3

Fig. 4-25. The state of the memory array, *M*, after each item in the reference string is processed. The distance string will be discussed in the next section.

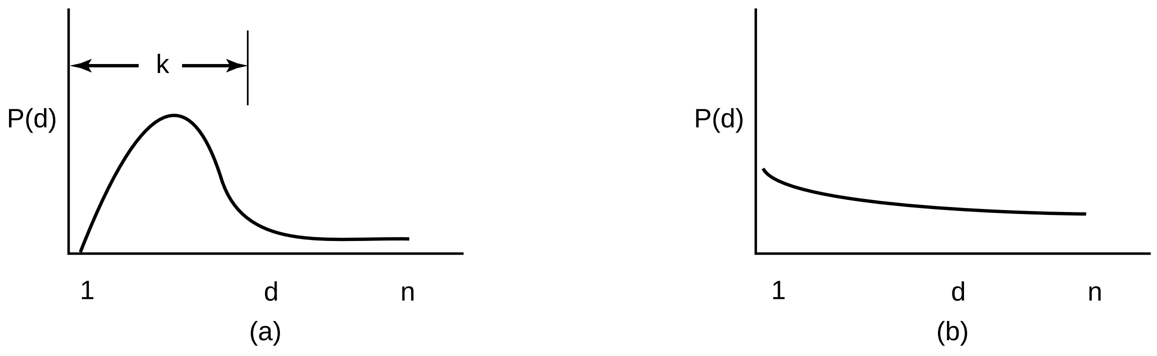
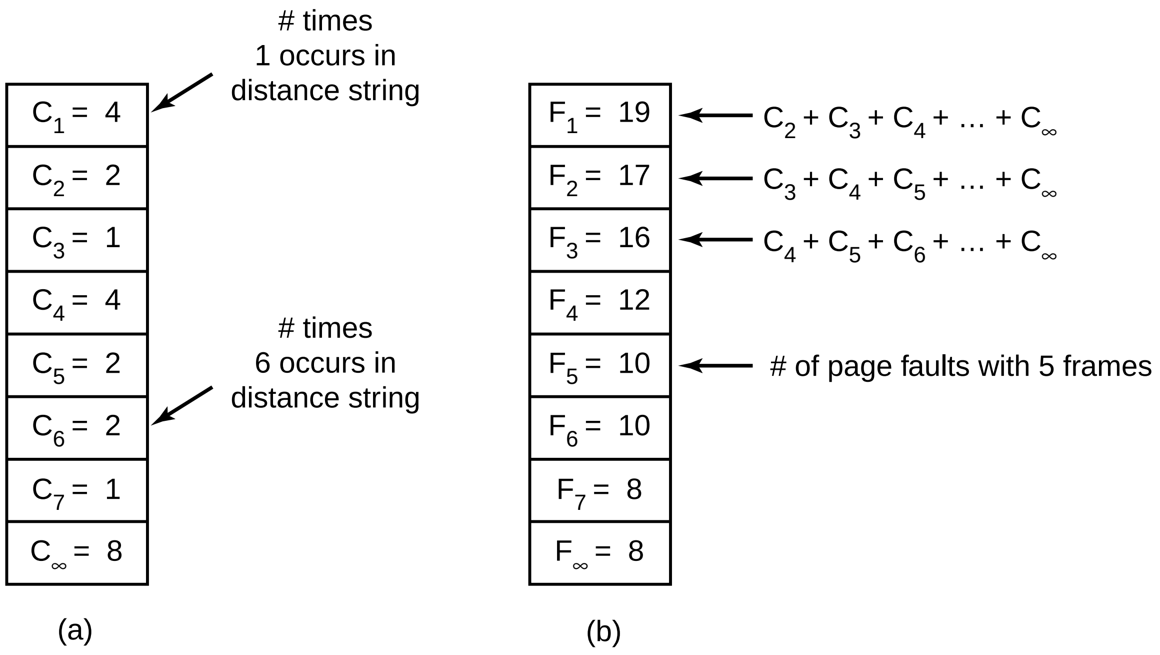


Fig. 4-26. Probability density functions for two hypothetical distance strings.



# times 1 occurs in distance string
# times 6 occurs in distance string
$C_1 = 4$
$C_2 = 2$
$C_3 = 1$
$C_4 = 4$
$C_5 = 2$
$C_6 = 2$
$C_7 = 1$
$C_\infty = 8$
(a)
$F_1 = 19$
$F_2 = 17$
$F_3 = 16$
$F_4 = 12$
$F_5 = 10$
$F_6 = 10$
$F_7 = 8$
$F_\infty = 8$
(b)
$C_2 + C_3 + C_4 + \dots + C_\infty$
$C_3 + C_4 + C_5 + \dots + C_\infty$
$C_4 + C_5 + C_6 + \dots + C_\infty$
# of page faults with 5 frames

Fig. 4-27. Computation of the page fault rate from the distance string. (a) The *C* vector. (b) *F* vector.

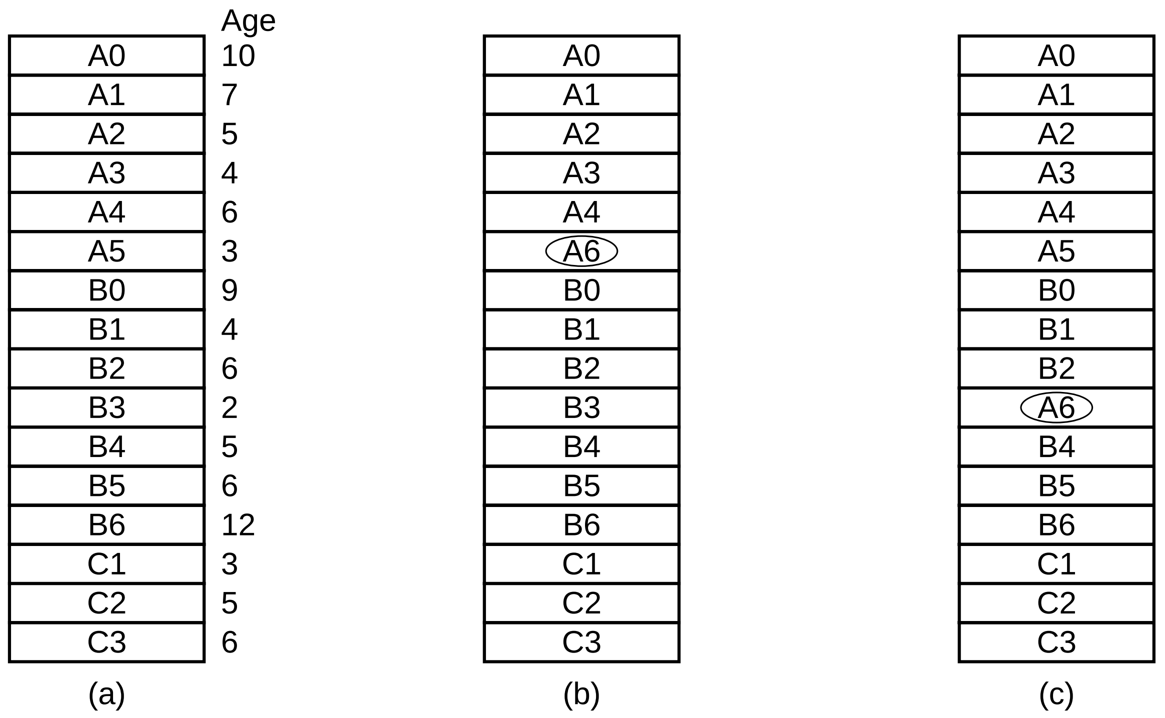


Fig. 4-28. Local versus global page replacement. (a) Original configuration. (b) Local page replacement. (c) Global page replacement.

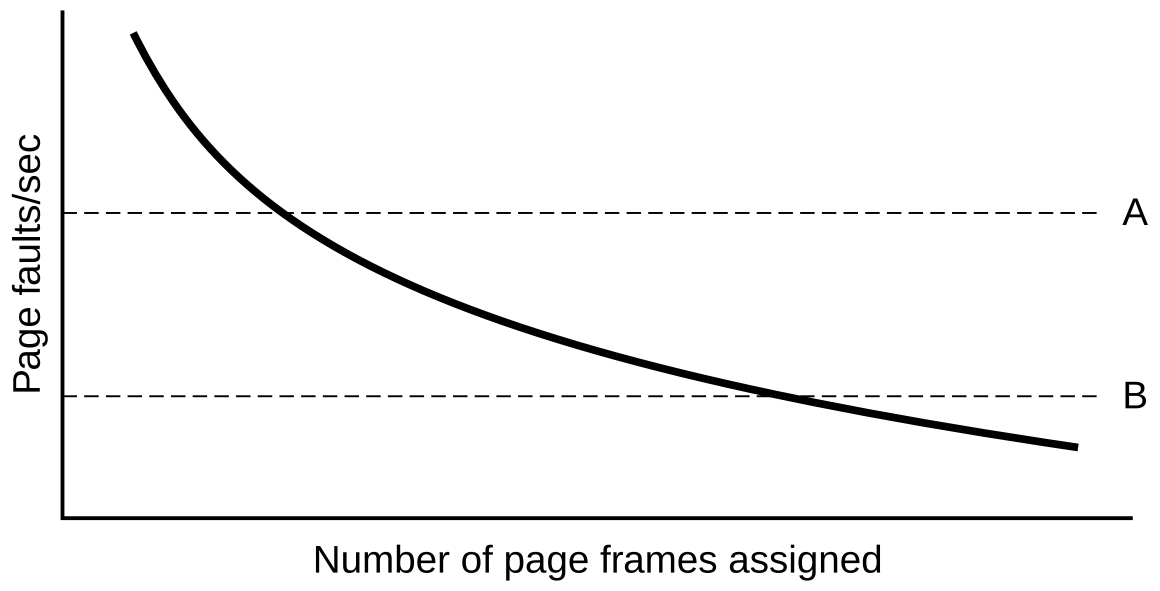


Fig. 4-29. Page fault rate as a function of the number of page frames assigned.

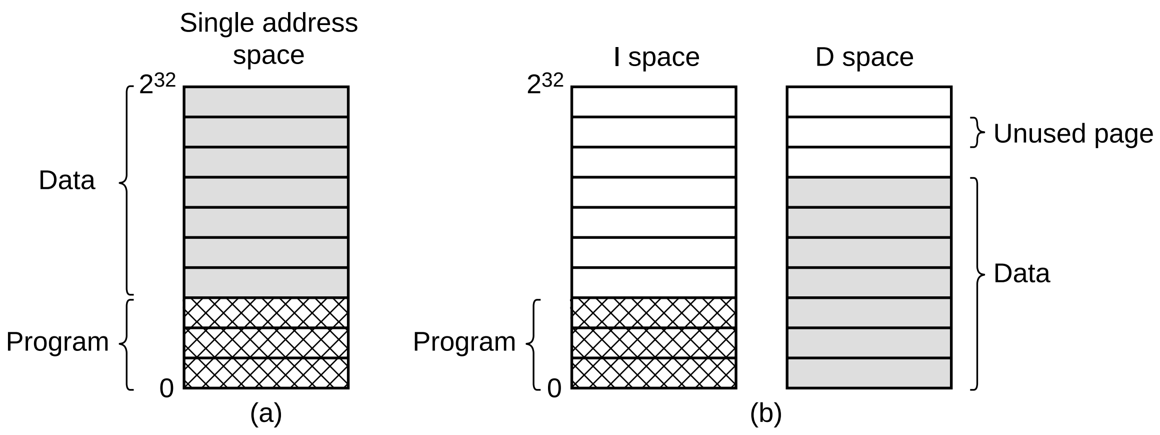


Fig. 4-30. (a) One address space. (b) Separate I and D spaces.

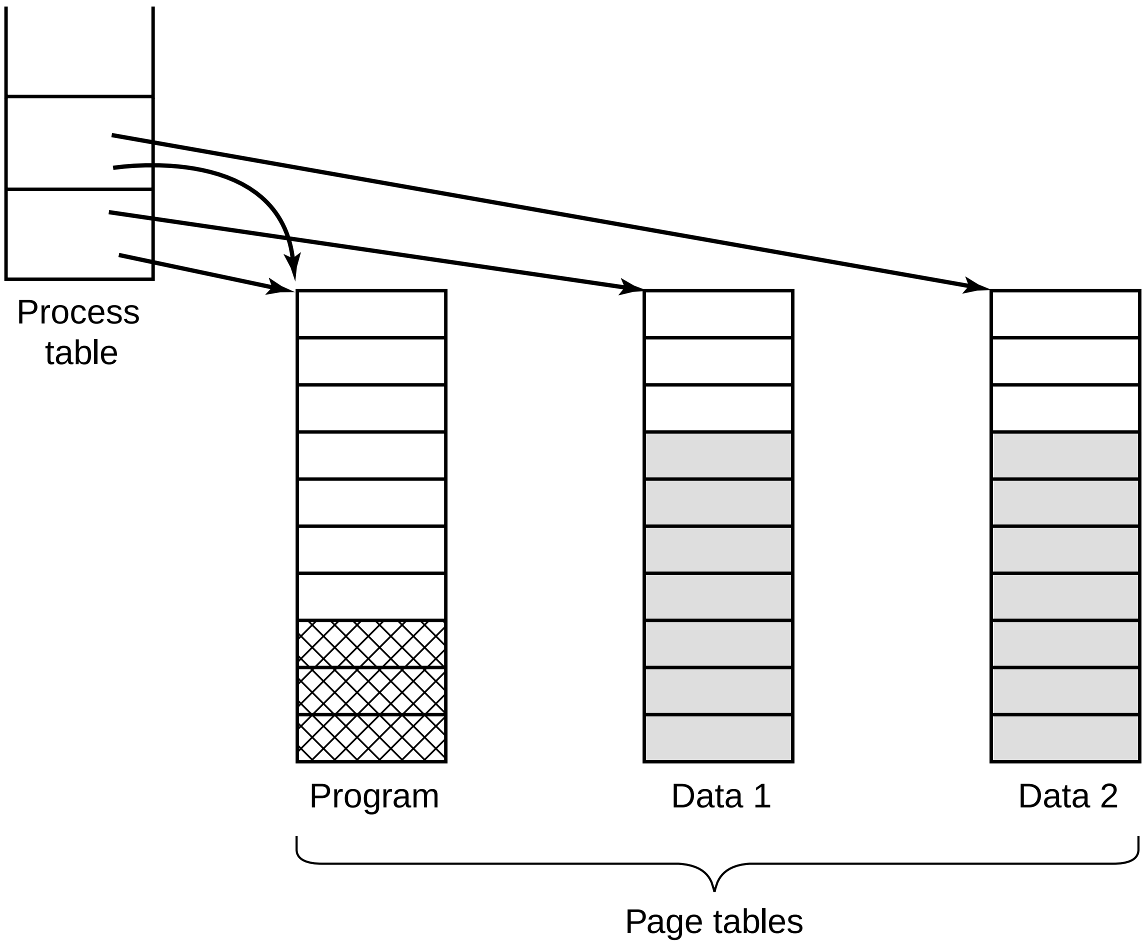


Fig. 4-31. Two processes sharing the same program sharing its page table.

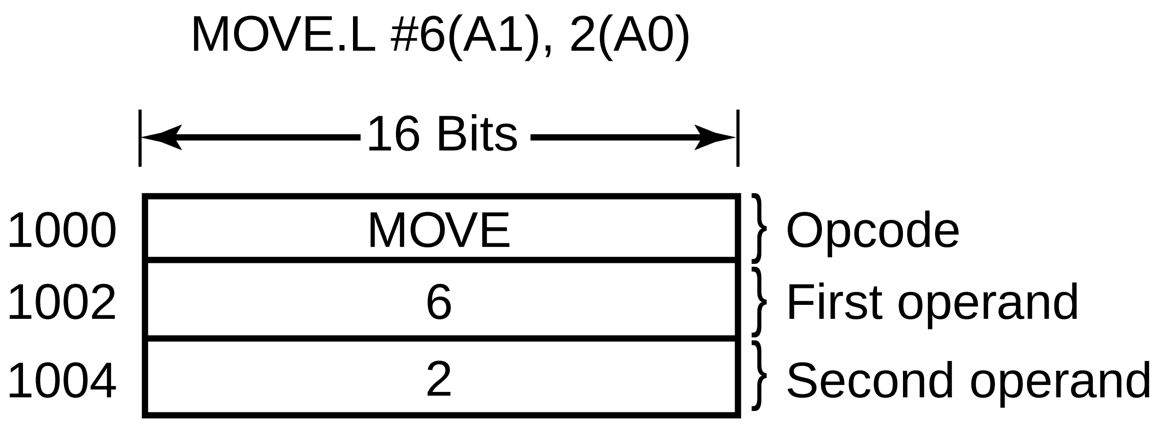


Fig. 4-32. An instruction causing a page fault.

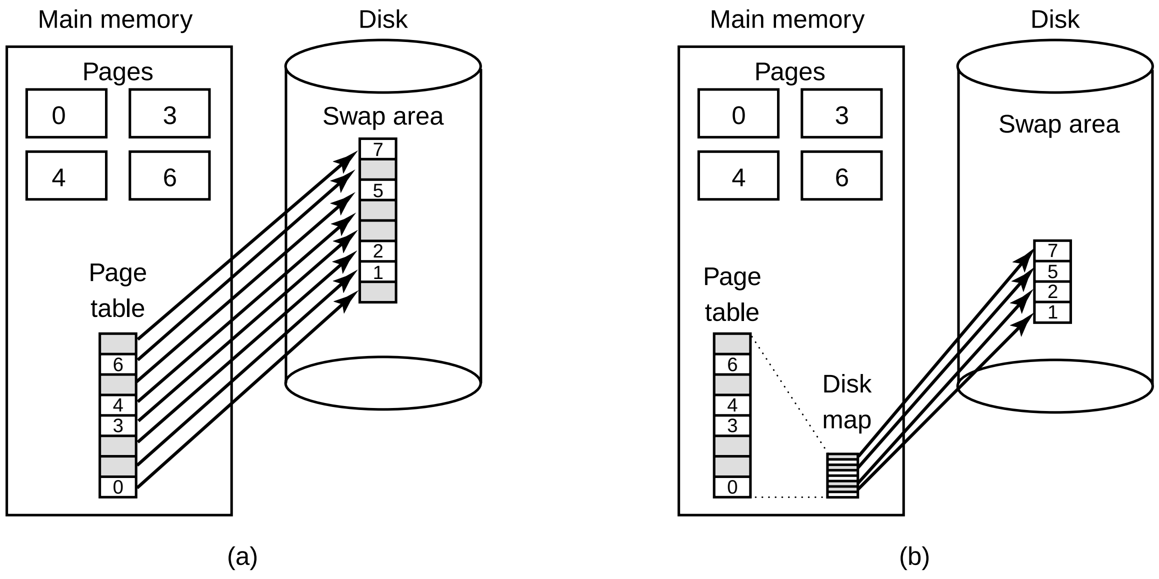


Fig. 4-33. (a) Paging to a static swap area. (b) Backing up pages dynamically.

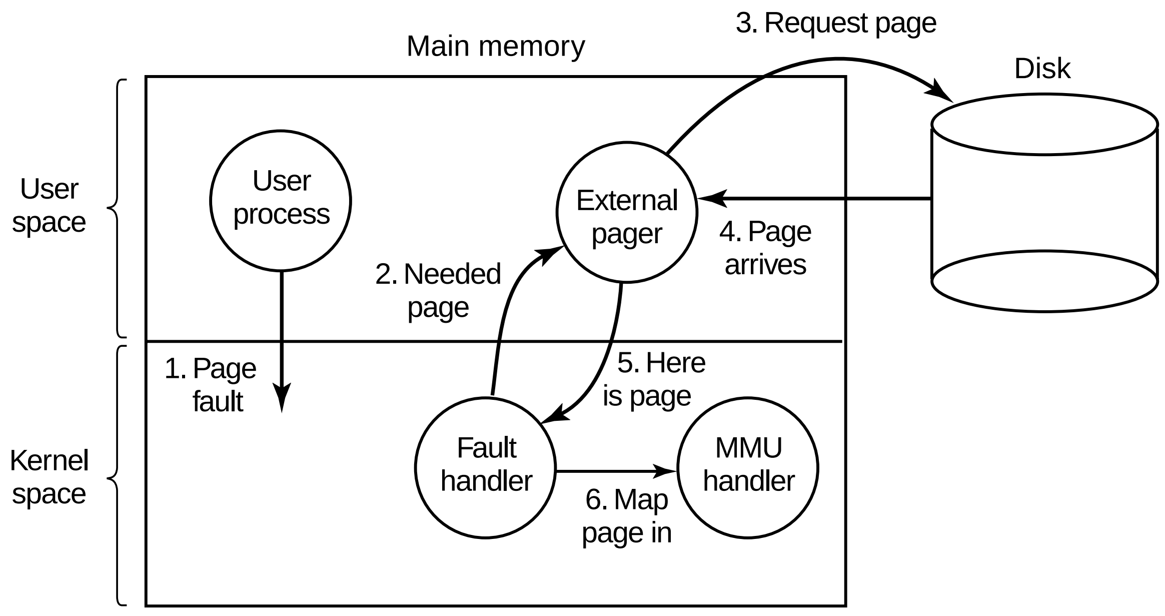


Fig. 4-34. Page fault handling with an external pager.

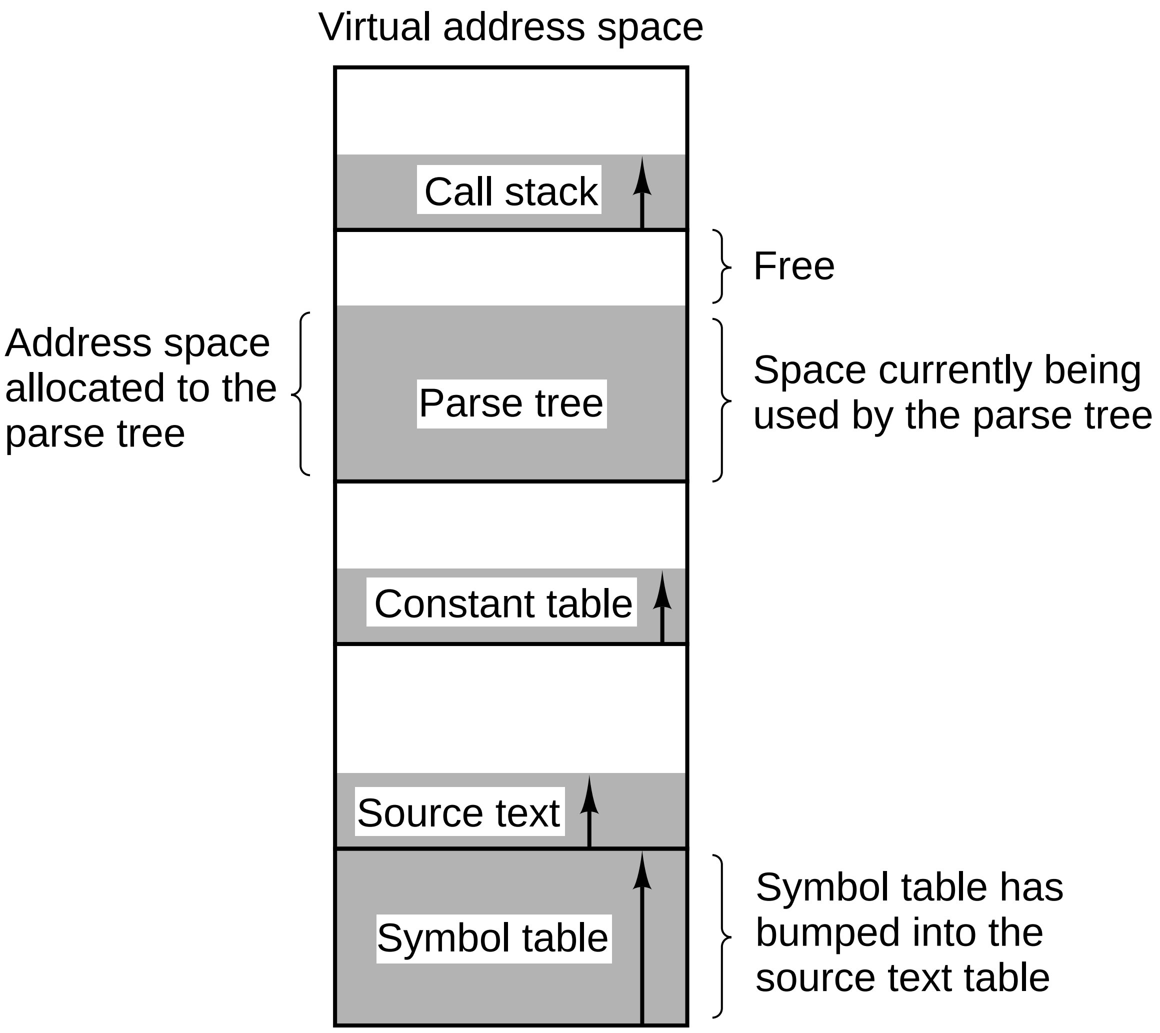

Virtual address space
Call stack
Free
Address space allocated to the parse tree
Parse tree
Space currently being used by the parse tree
Constant table
Source text
Symbol table
Symbol table has bumped into the source text table

Fig. 4-35. In a one-dimensional address space with growing tables, one table may bump into another.

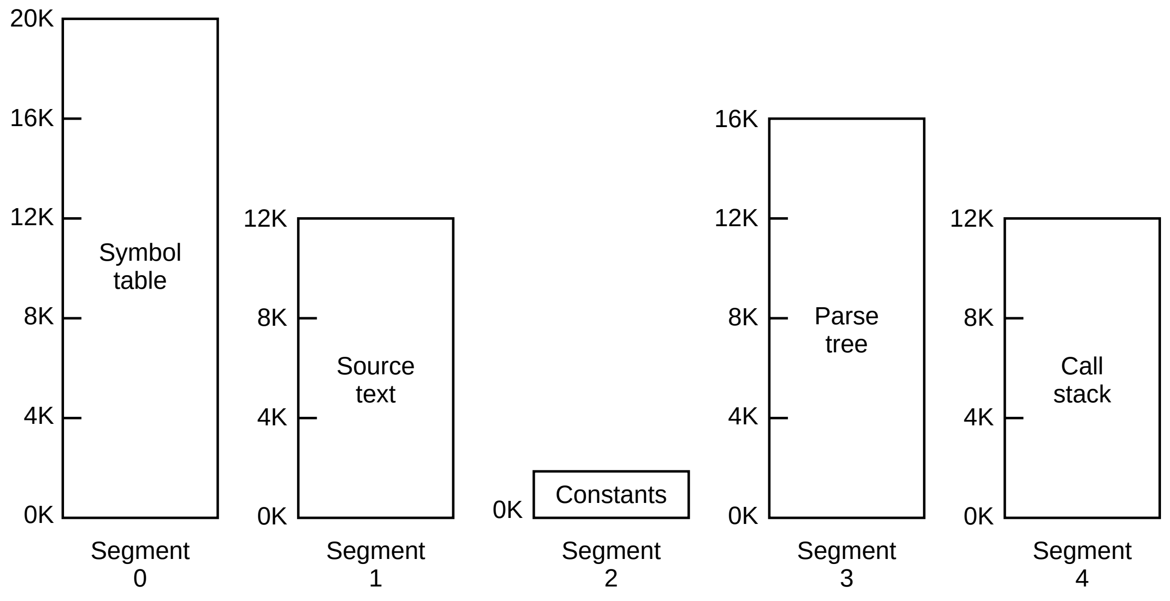


Fig. 4-36. A segmented memory allows each table to grow or shrink independently of the other tables.

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

Fig. 4-37. Comparison of paging and segmentation.

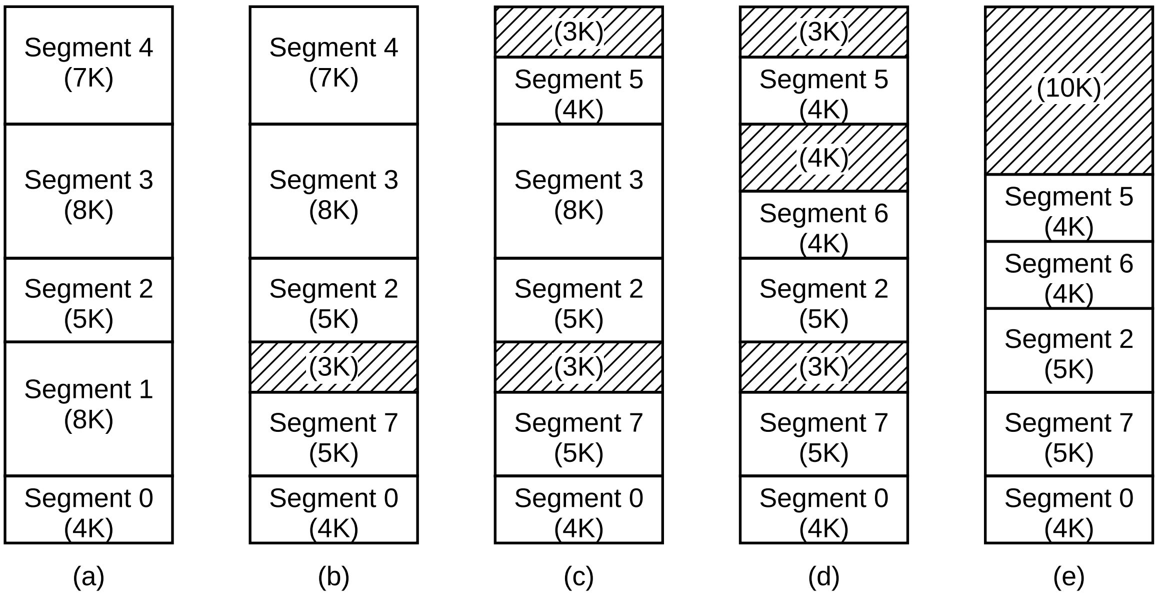


Fig. 4-38. (a)-(d) Development of checkerboarding. (e) Removal of the checkerboarding by compaction.

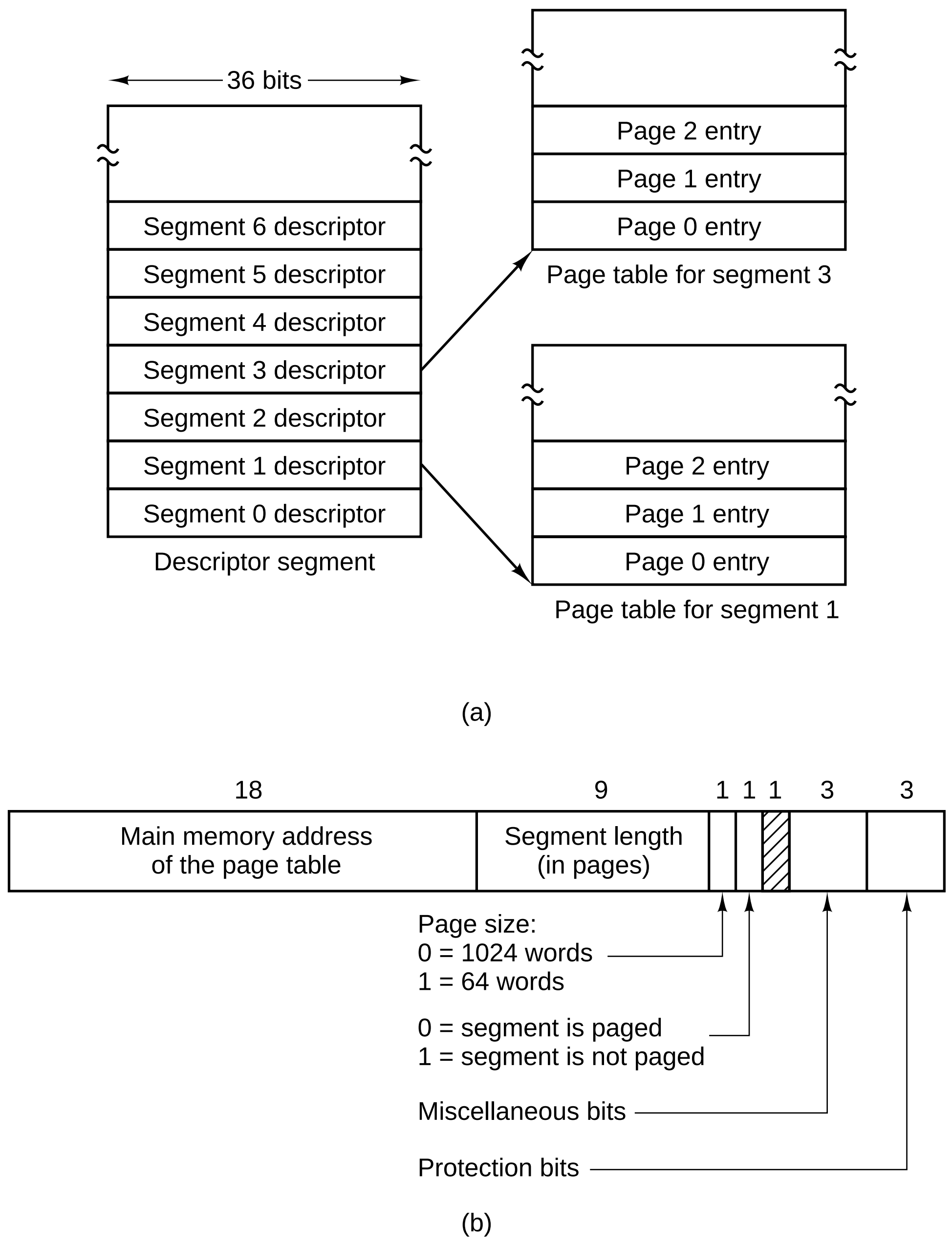


Fig. 4-39. The MULTICS virtual memory. (a) The descriptor segment points to the page tables. (b) A segment descriptor. The numbers are the field lengths.

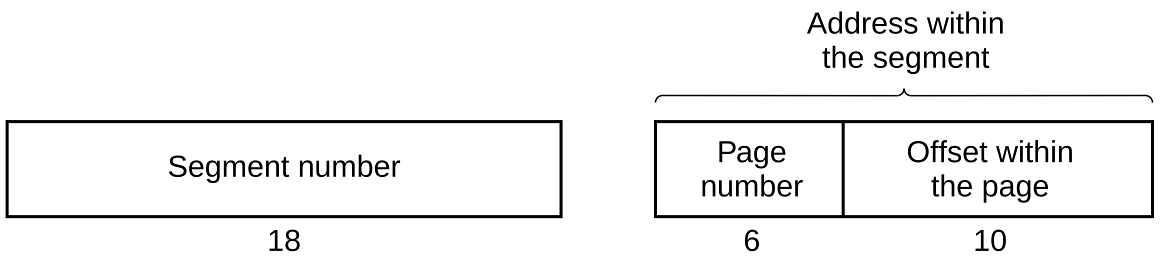


Fig. 4-40. A 34-bit MULTICS virtual address.

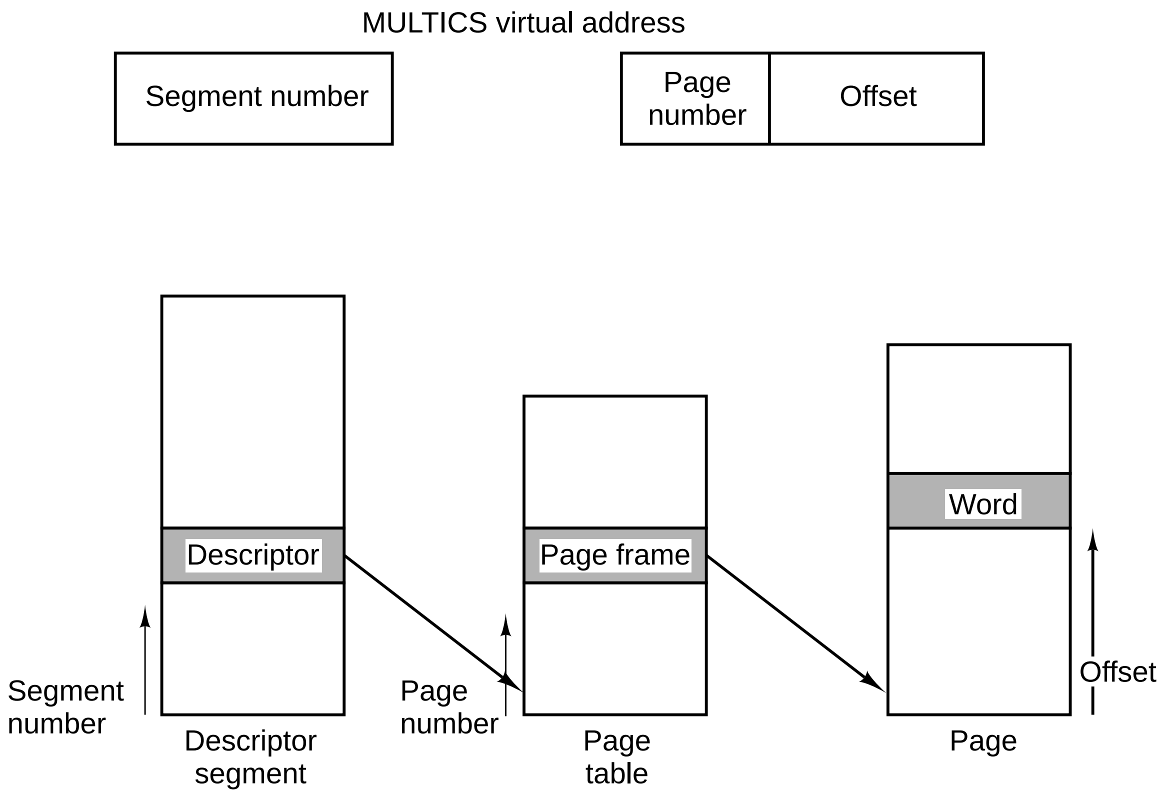


Fig. 4-41. Conversion of a two-part MULTICS address into a main memory address.

Comparison field					
Segment number	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

Fig. 4-42. A simplified version of the MULTICS TLB. The existence of two page sizes makes the actual TLB more complicated.

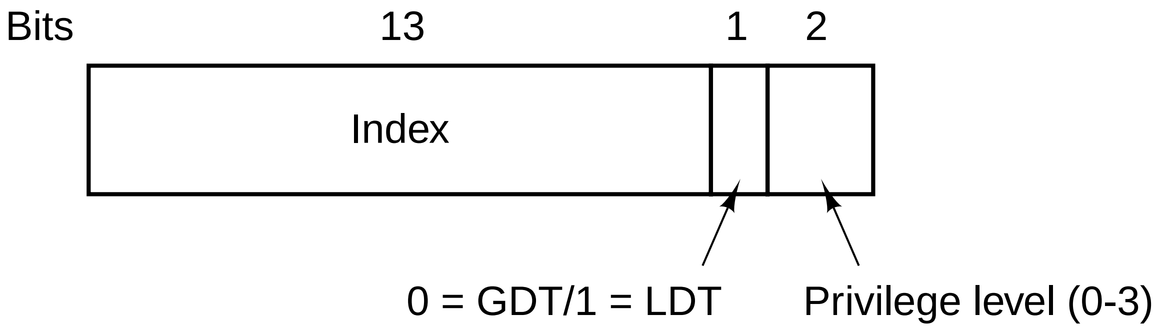


Fig. 4-43. A Pentium selector.

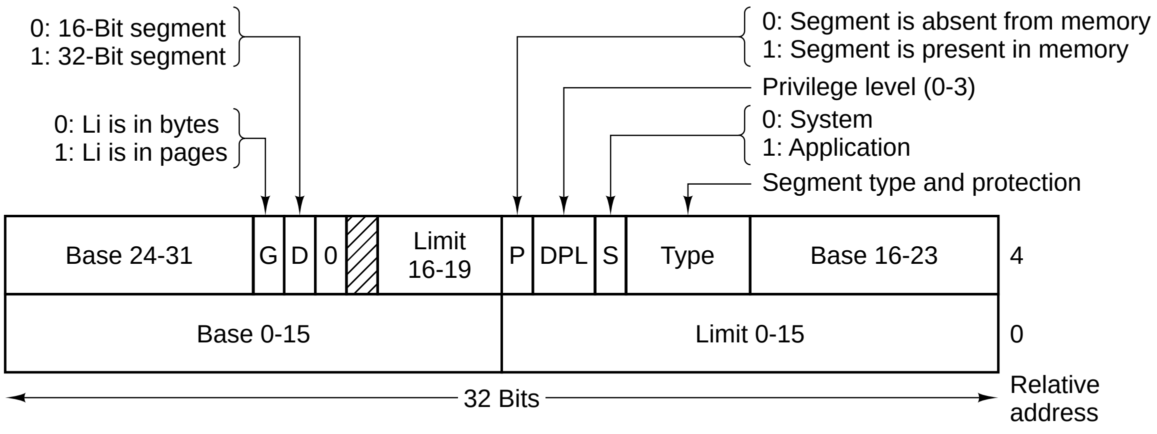


Fig. 4-44. Pentium code segment descriptor. Data segments differ slightly.

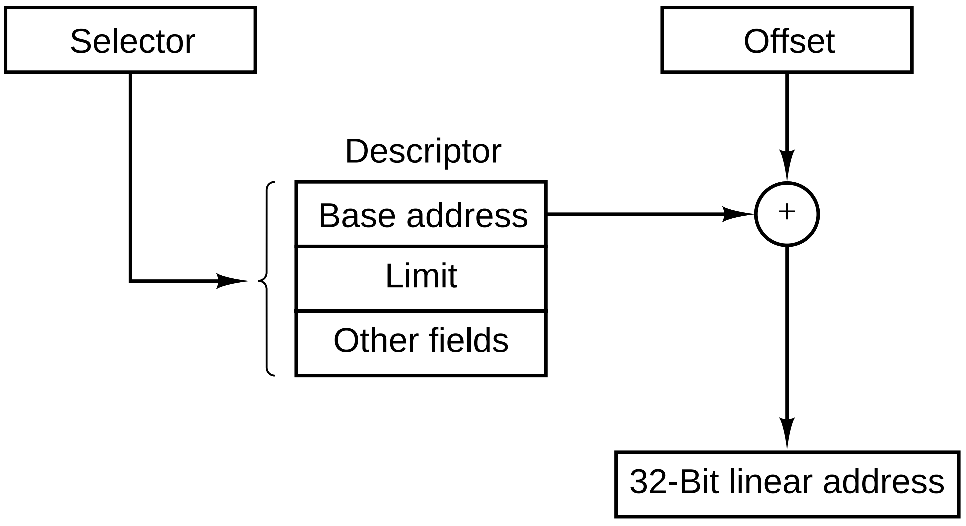


Fig. 4-45. Conversion of a (selector, offset) pair to a linear address.

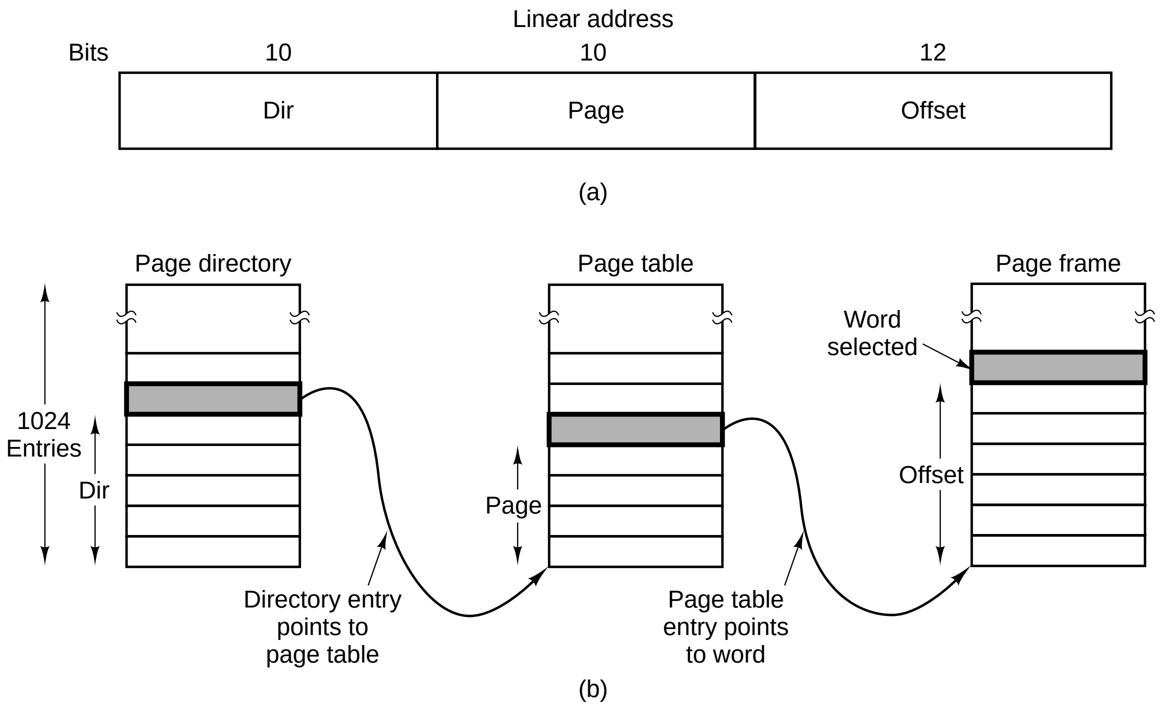


Fig. 4-46. Mapping of a linear address onto a physical address.

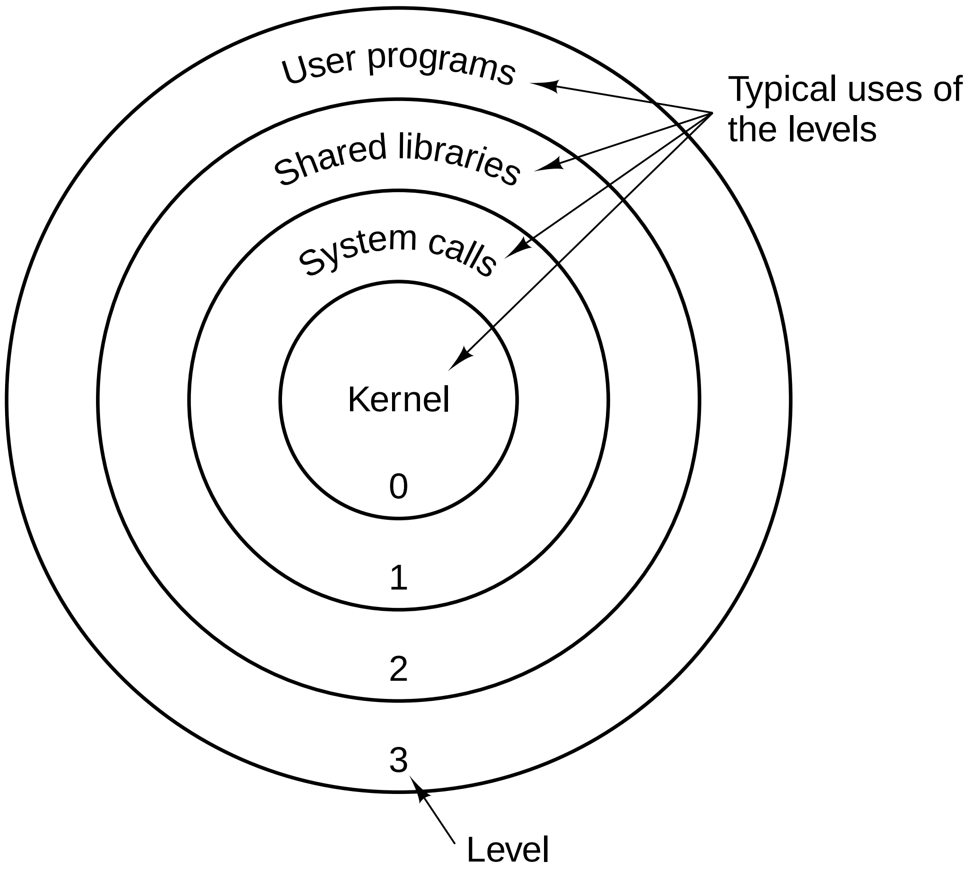


Fig. 4-47. Protection on the Pentium.