MODERN OPERATING SYSTEMS

Third Edition

ANDREW S. TANENBAUM

Chapter 3
Memory Management

No Memory Abstraction

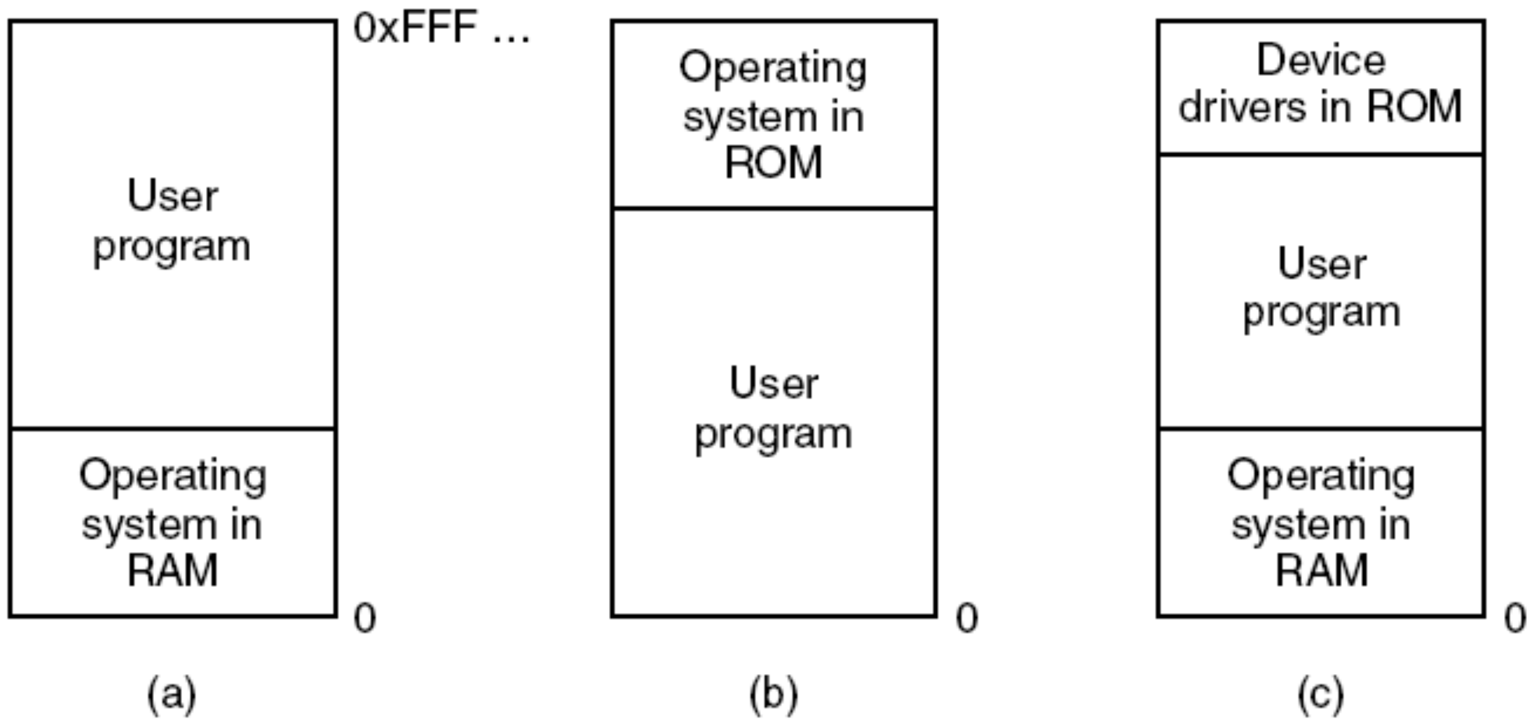


Figure 3-1. Three simple ways of organizing memory with an operating system and one user process. Other possibilities also exist.

Multiple Programs Without Memory Abstraction (1)

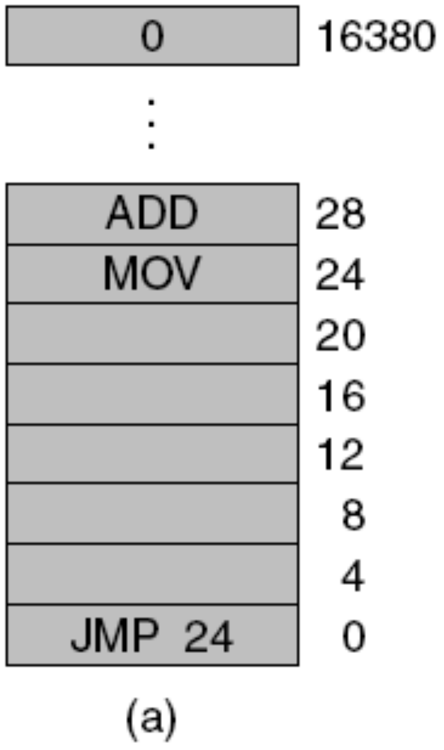


Figure 3-2. Illustration of the relocation problem. (a) A 16-KB program.

Multiple Programs Without Memory Abstraction (2)

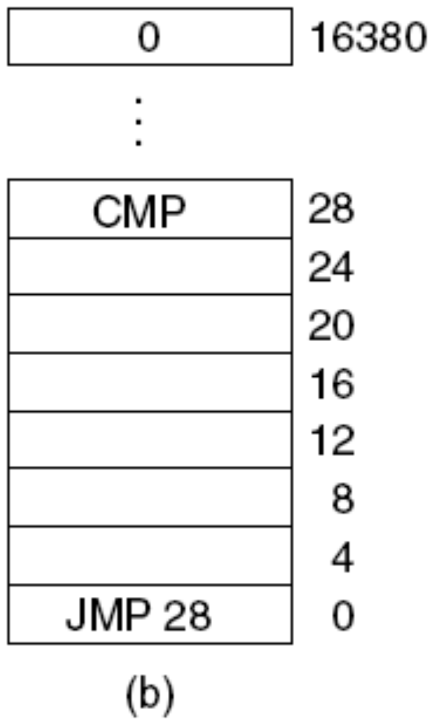


Figure 3-2. Illustration of the relocation problem. (b) Another 16-KB program.

Multiple Programs Without Memory Abstraction (3)

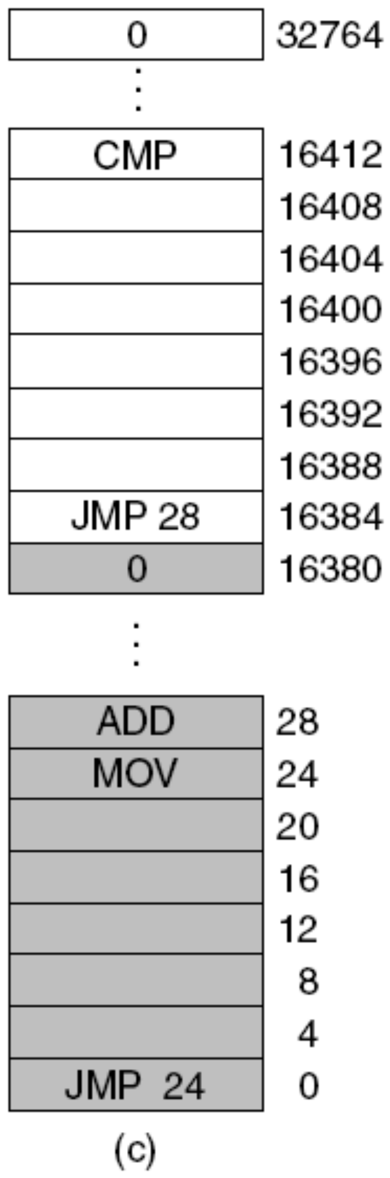


Figure 3-2. Illustration of the relocation problem. (c) The two programs loaded consecutively into memory.

Base and Limit Registers

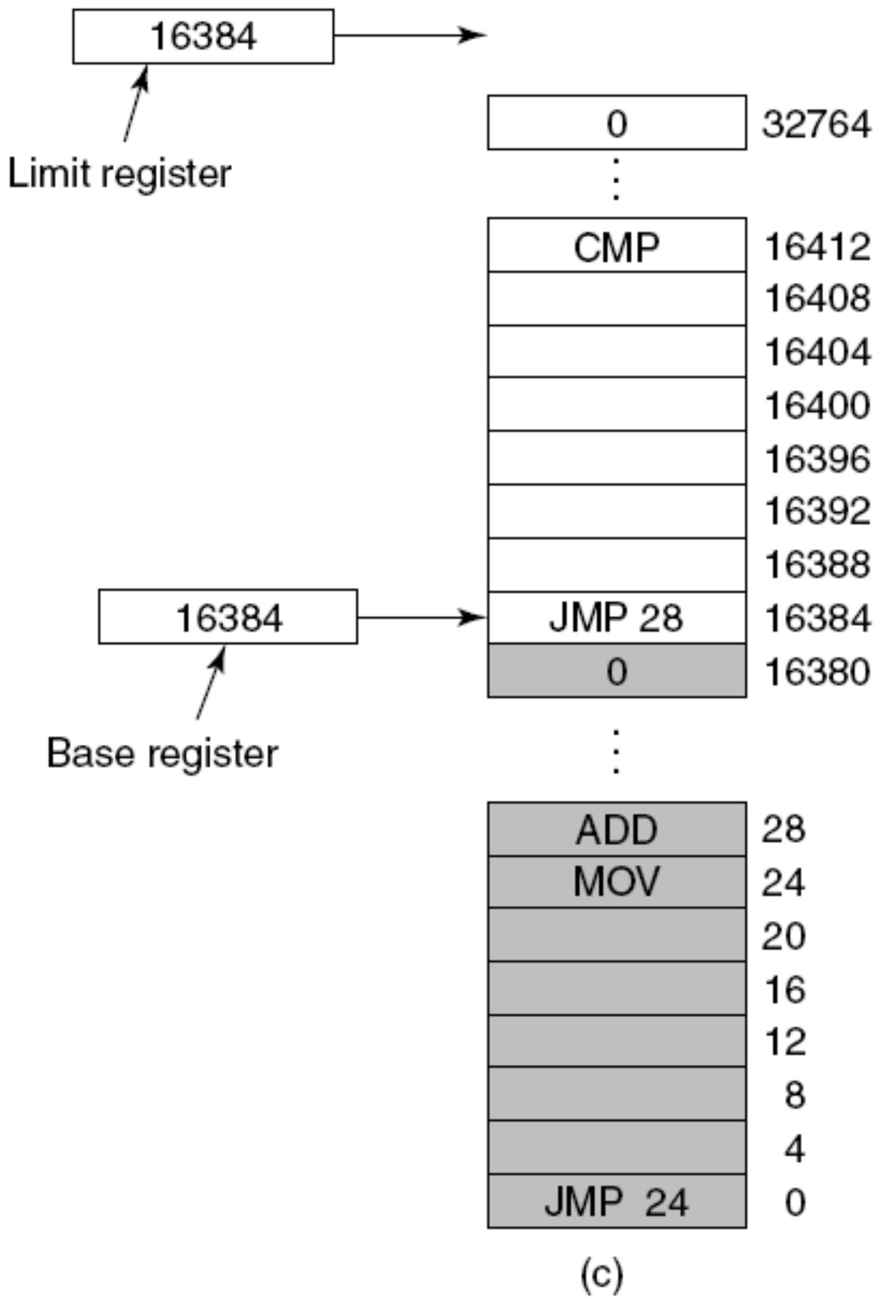


Figure 3-3. Base and limit registers can be used to give each process a separate address space.

Swapping (1)

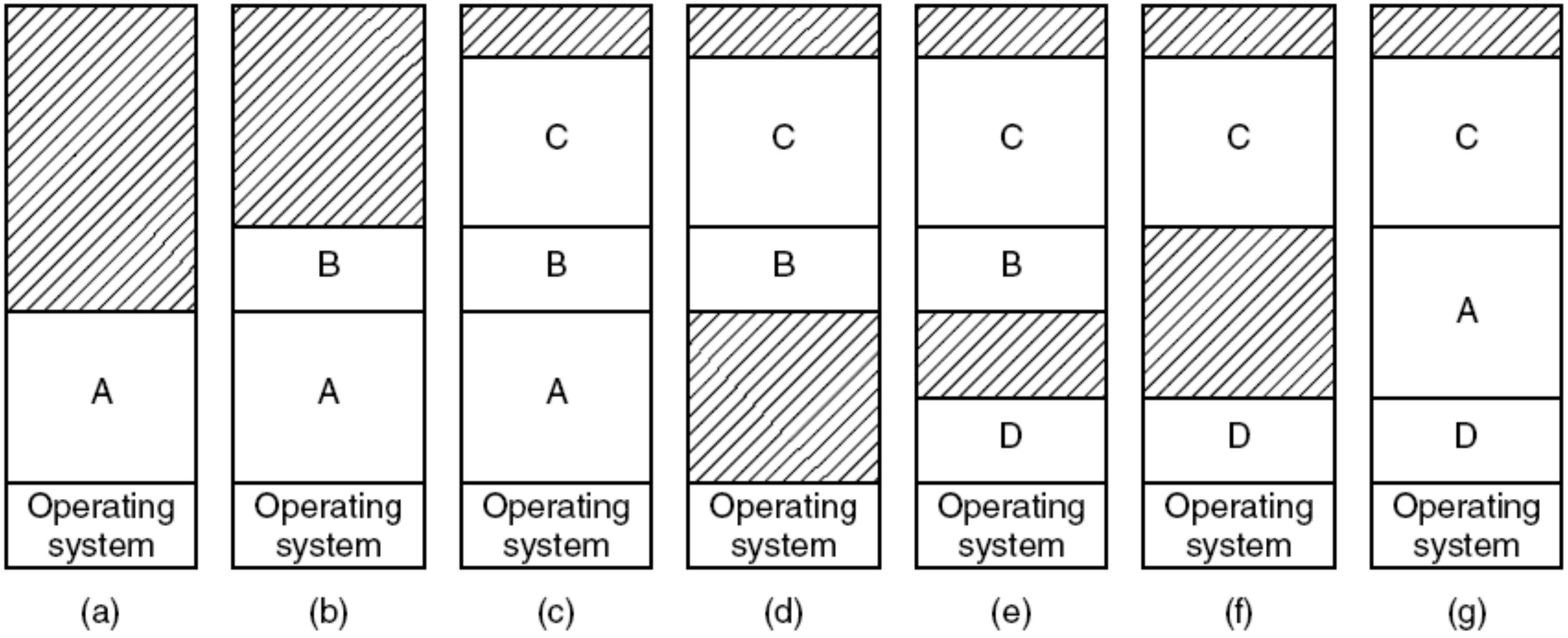


Figure 3-4. Memory allocation changes as processes come into memory and leave it. The shaded regions are unused memory.

Swapping (2)

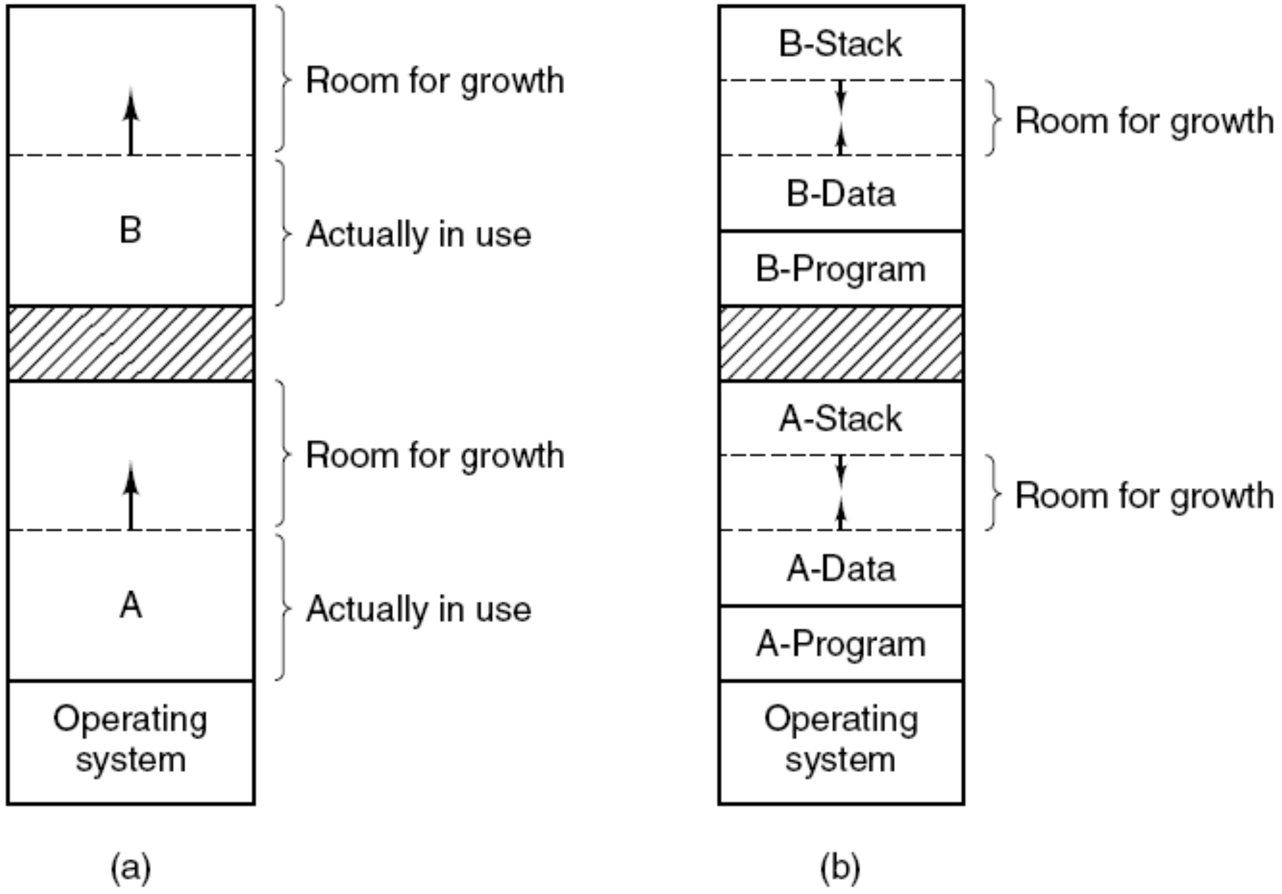


Figure 3-5. (a) Allocating space for growing data segment. (b) Allocating space for growing stack, growing data segment.

Memory Management with Bitmaps

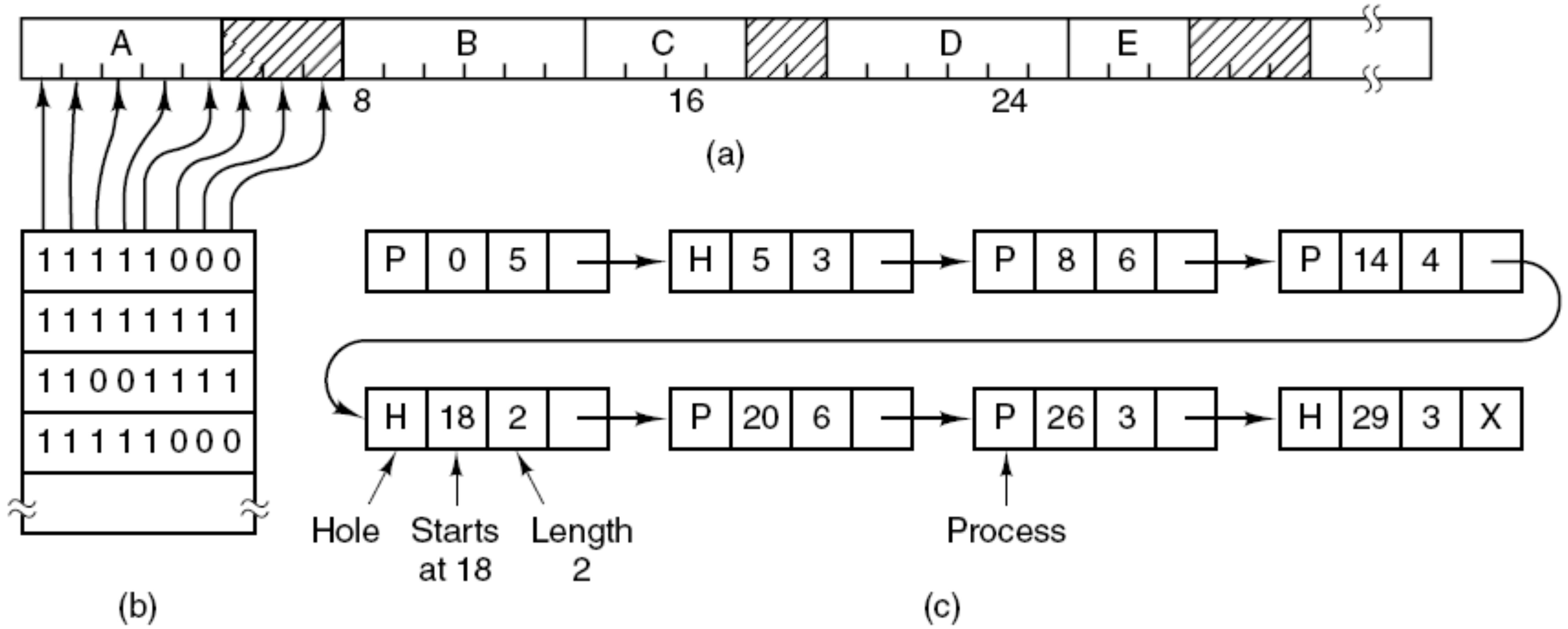


Figure 3-6. (a) A part of memory with five processes and three holes. The tick marks show the memory allocation units. The shaded regions (0 in the bitmap) are free. (b) The corresponding bitmap. (c) The same information as a list.

Memory Management with Linked Lists

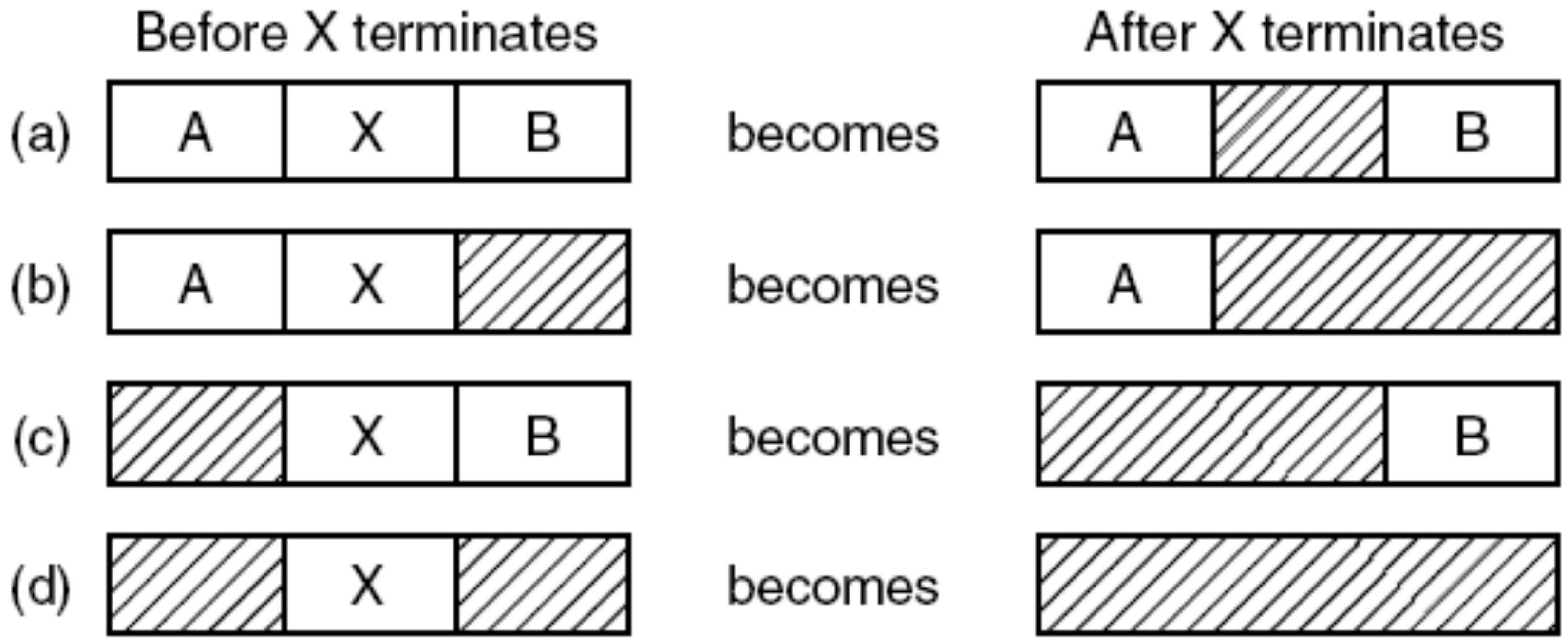


Figure 3-7. Four neighbor combinations for the terminating process, X.

Virtual Memory – Paging (1)

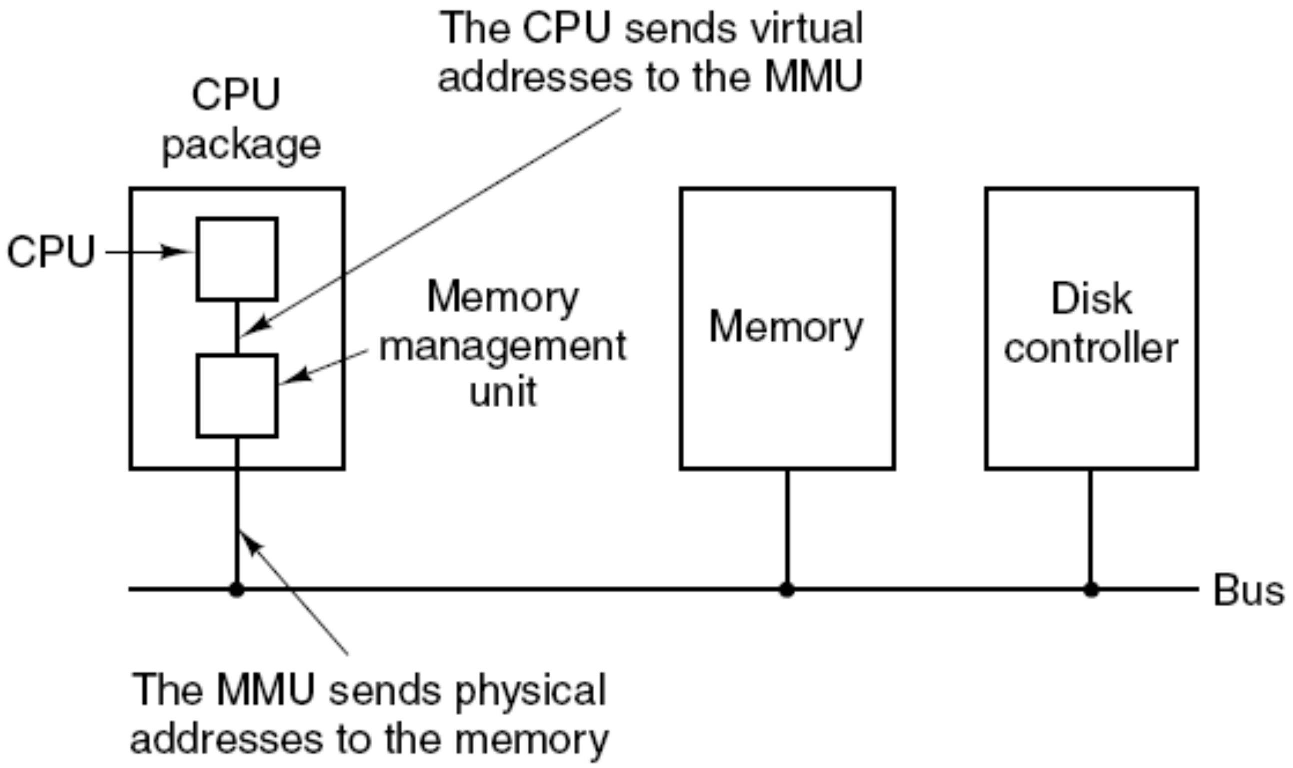
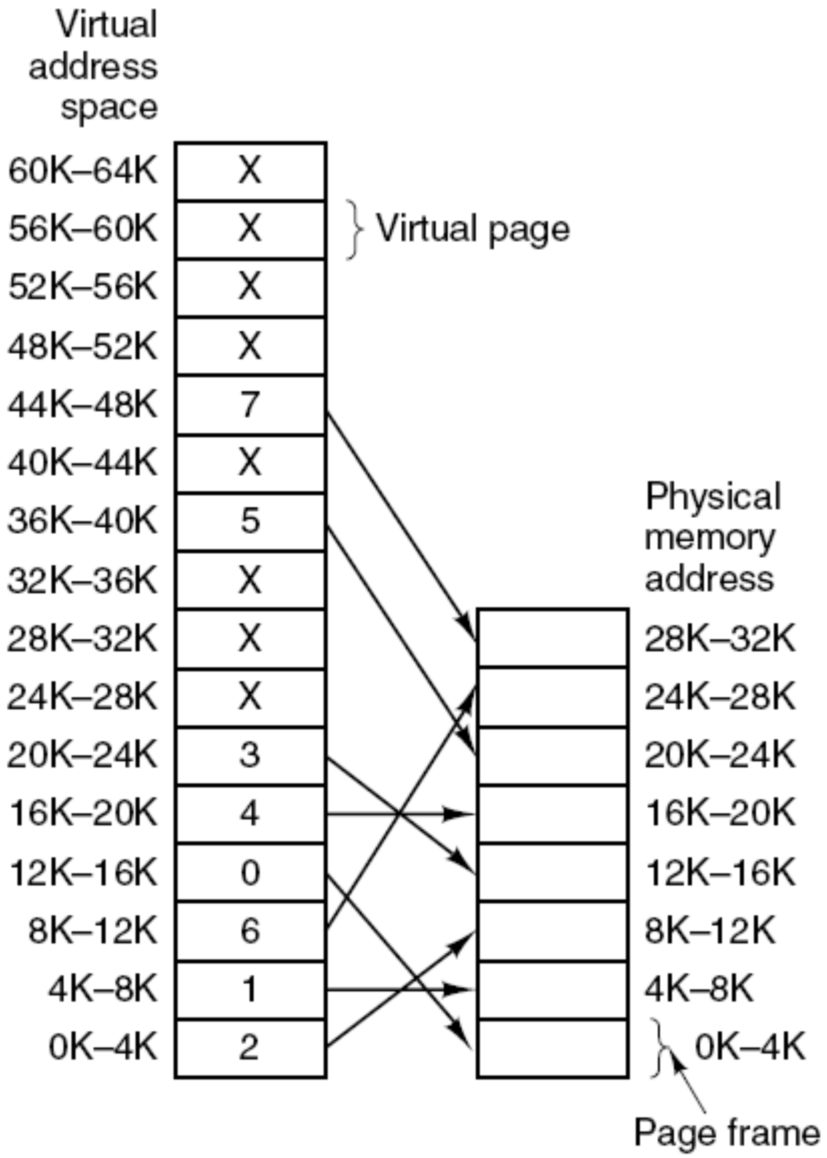

The CPU sends virtual addresses to the MMU
CPU package
CPU
Memory management unit
Memory
Disk controller
Bus
The MMU sends physical addresses to the memory

Figure 3-8. The position and function of the MMU – shown as being a part of the CPU chip (it commonly is nowadays). Logically it could be a separate chip, was in years gone by.

Paging (2)

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



Paging (3)

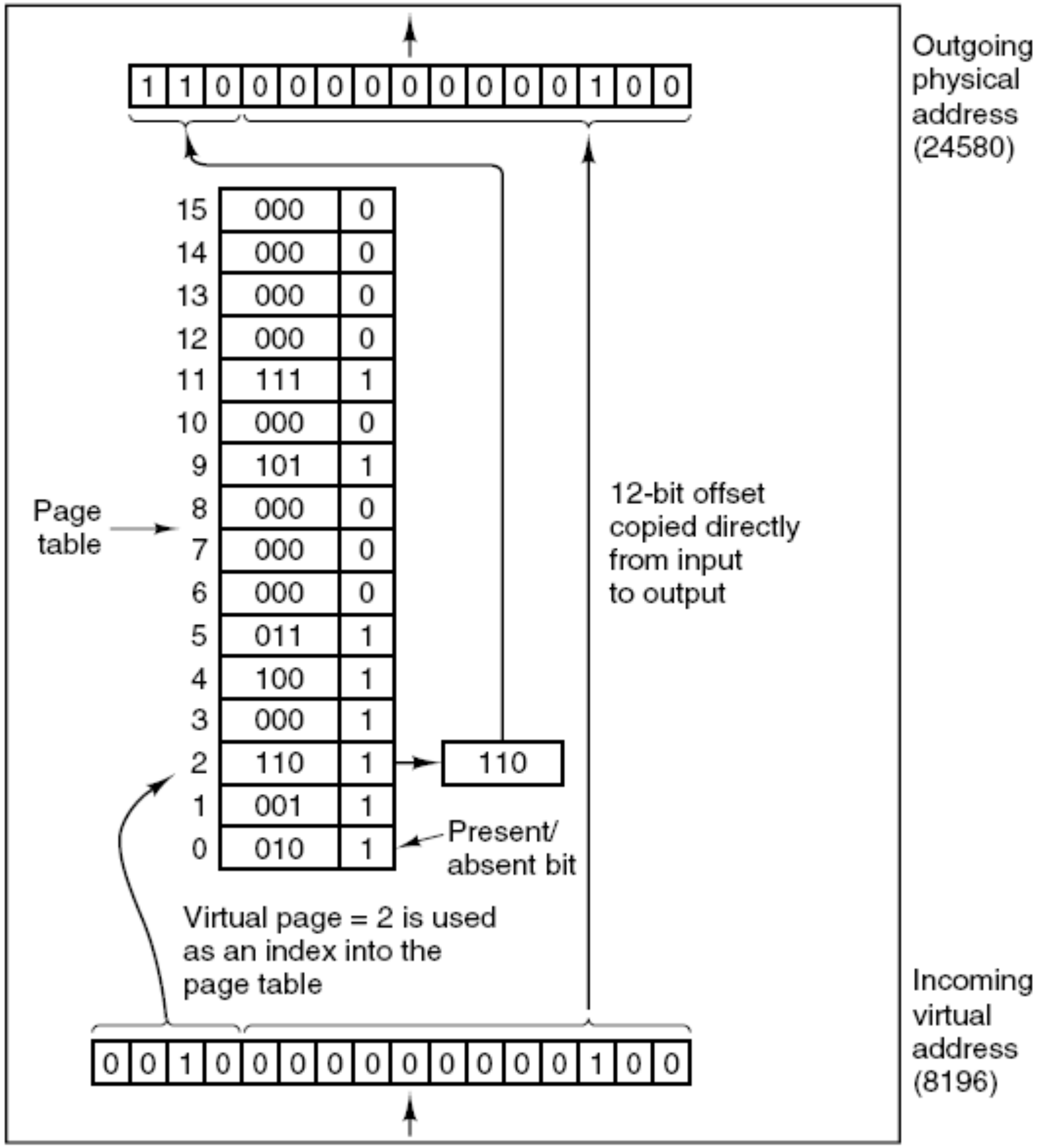


Figure 3-10. The internal operation of the MMU with 16 4-KB pages.

Structure of Page Table Entry

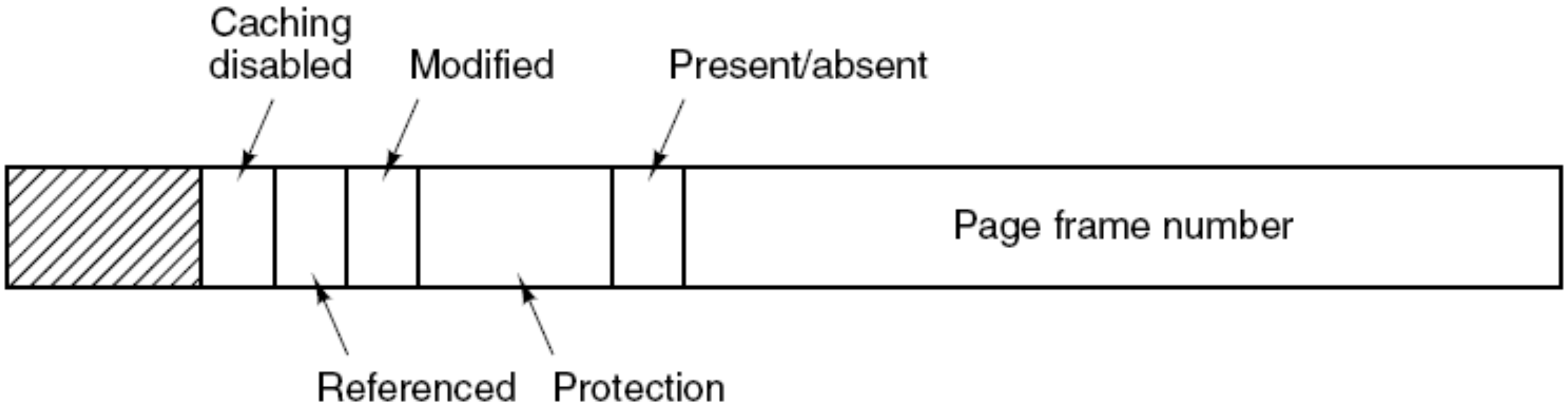


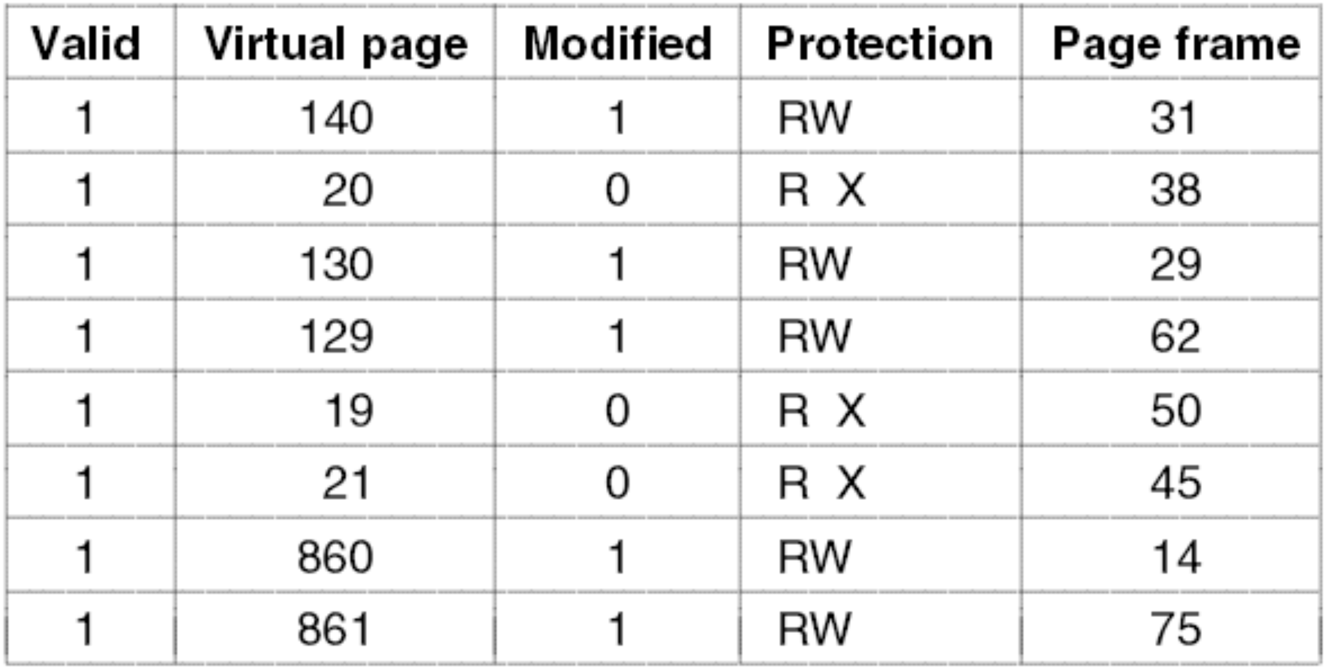
Figure 3-11. A typical page table entry.

Speeding Up Paging

Paging implementation issues

1. The mapping from virtual address to physical address must be fast.
2. 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

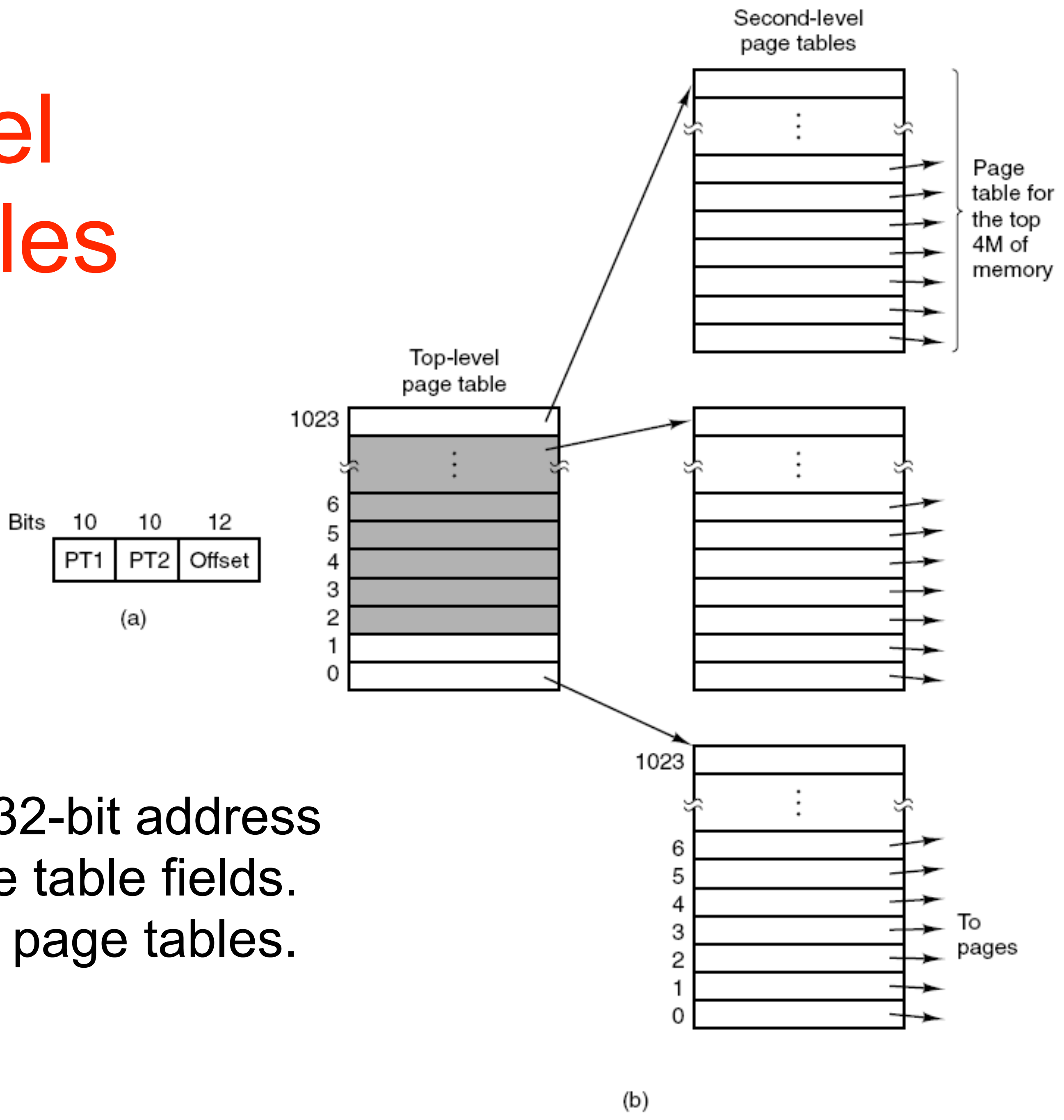


Figure 3-13. (a) A 32-bit address with two page table fields. (b) Two-level page tables.

Inverted Page Tables

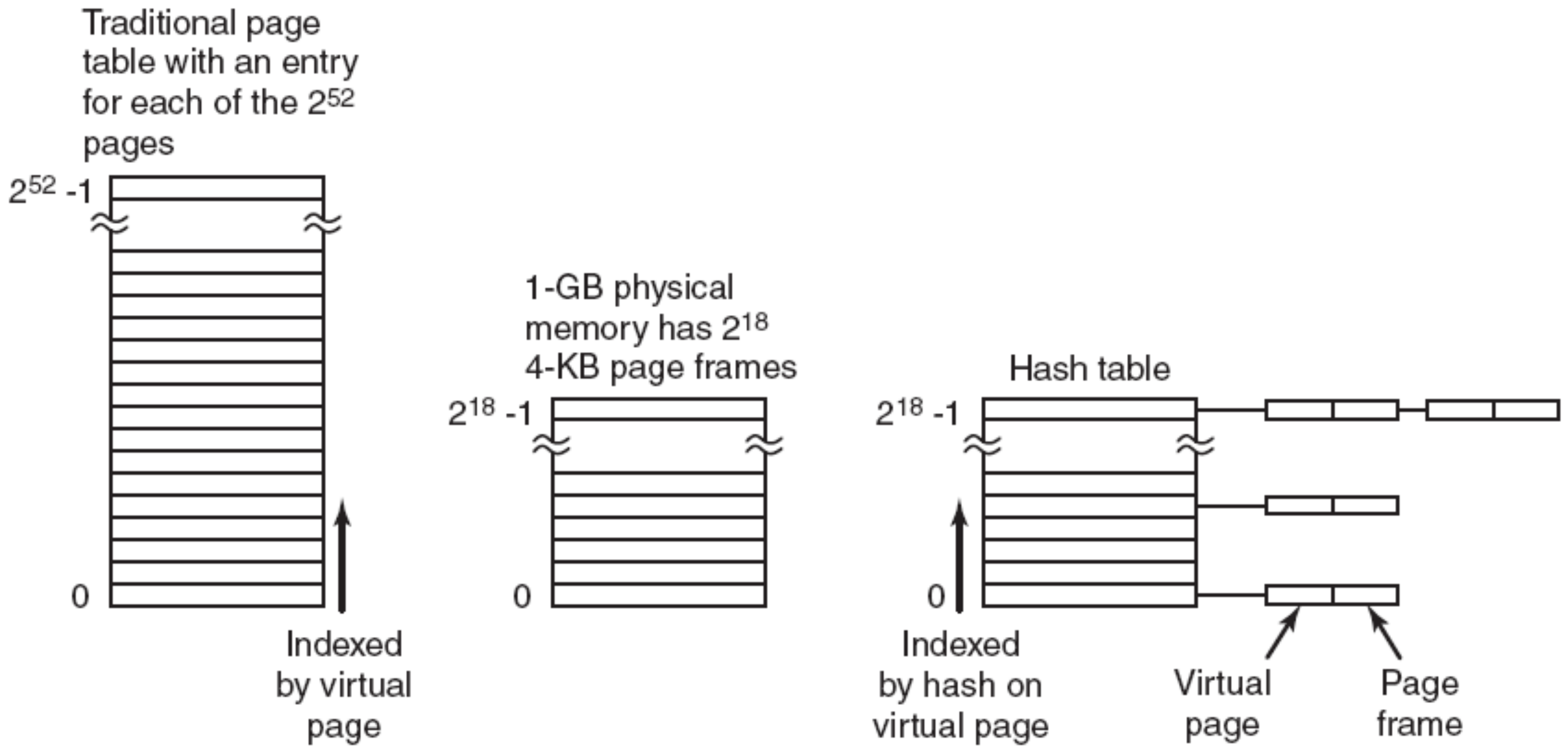

Traditional page table with an entry for each of the 2^{52} pages
2^{52} -1
0
Indexed by virtual page
1-GB physical memory has 2^{18} 4-KB page frames
2^{18} -1
0
Hash table
2^{18} -1
0
Indexed by hash on virtual page
Virtual page
Page frame

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

Page Replacement Algorithms

- Optimal page replacement algorithm
- Not recently used page replacement
- First-In, First-Out page replacement
- Second chance page replacement
- Clock page replacement
- Least recently used page replacement
- Working set page replacement
- WSClock page replacement

Second Chance Algorithm

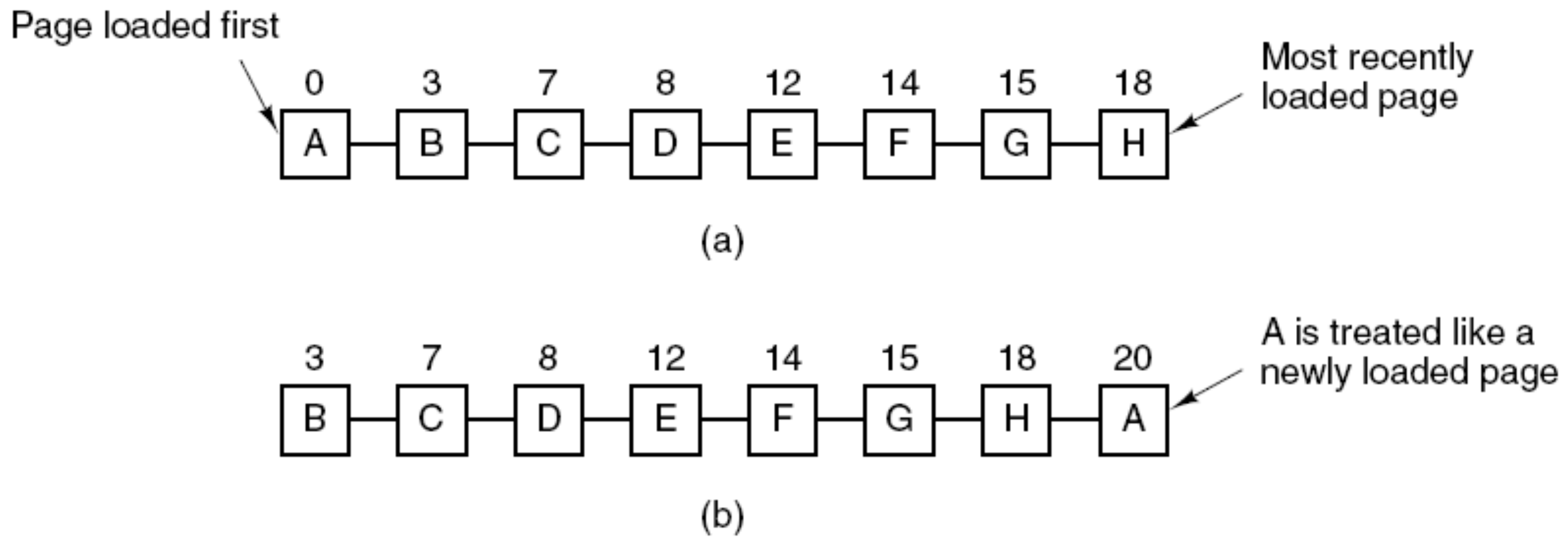
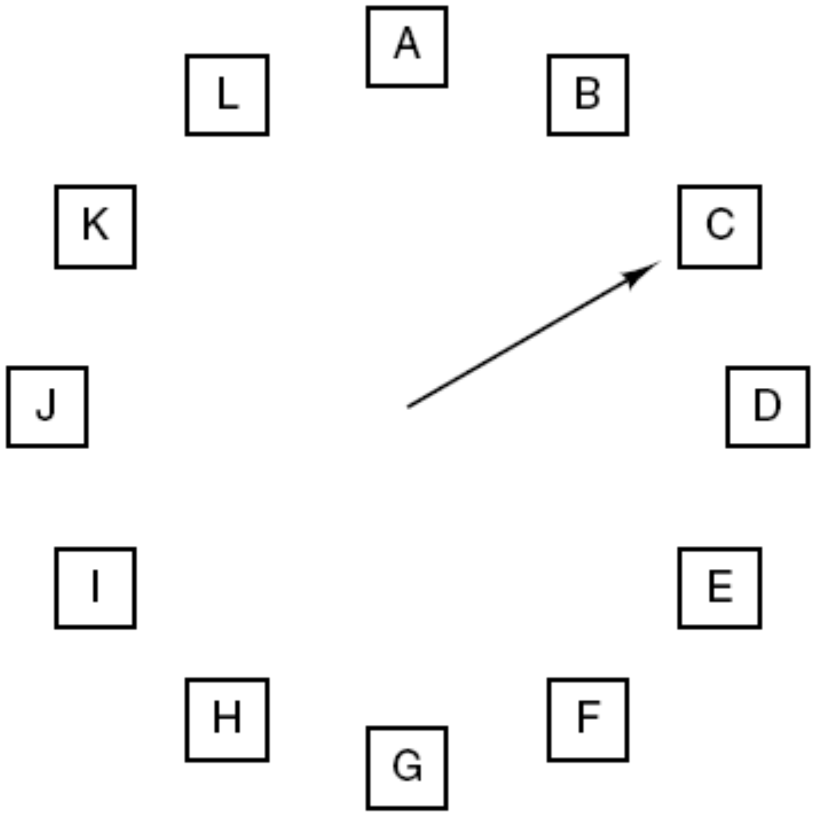


Figure 3-15. Operation of second chance.
(a) Pages sorted in FIFO order.
(b) Page list if a page fault occurs at time 20 and A has its R bit set. The numbers above the pages are their load times.

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

Figure 3-16. The clock page replacement algorithm.

LRU Page Replacement Algorithm

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

(a)

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

(b)

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

(c)

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

(d)

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

(e)

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

(f)

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

(g)

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

(h)

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

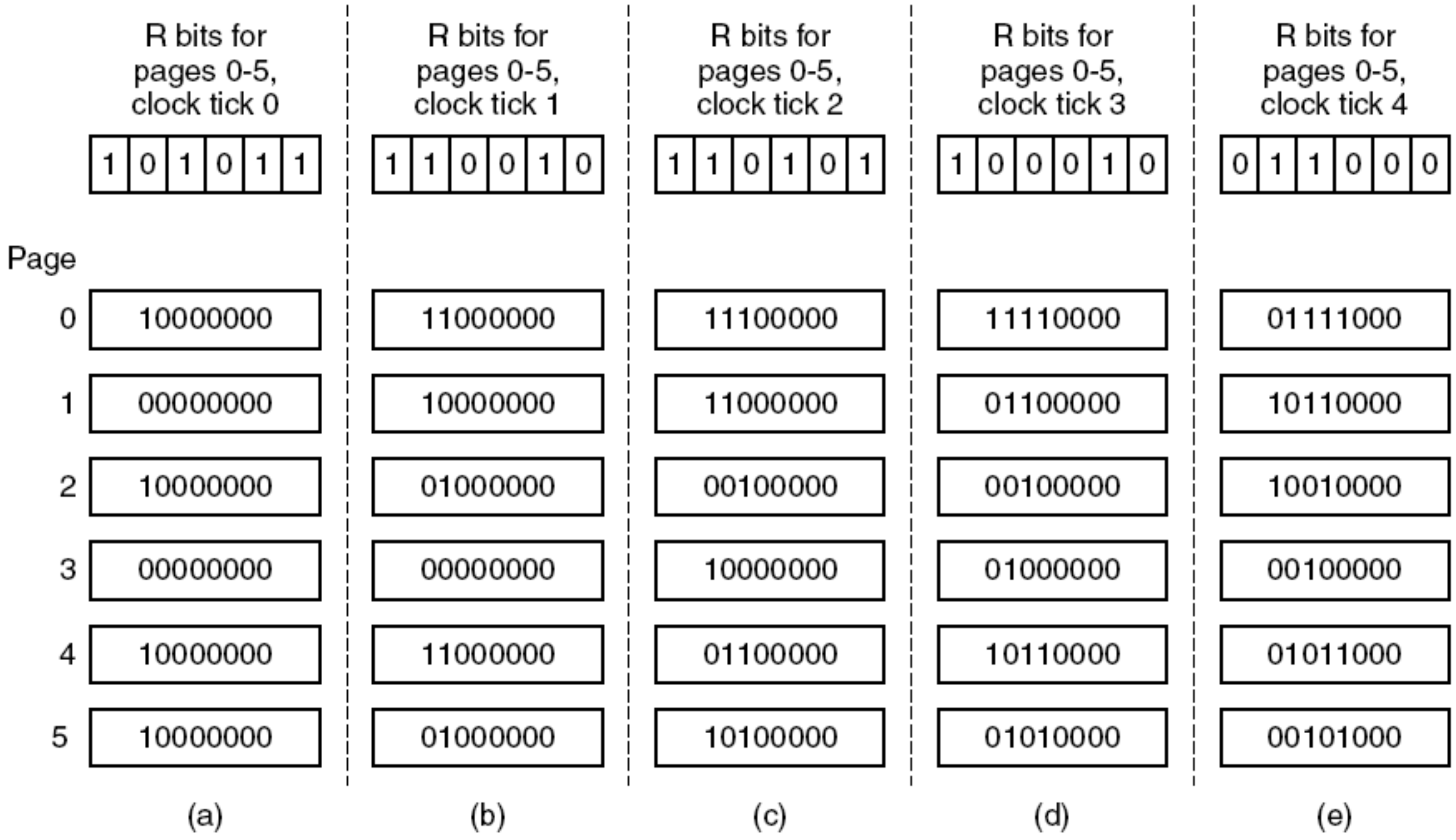
(i)

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

(j)

Figure 3-17. LRU using a matrix when pages are referenced in the order 0, 1, 2, 3, 2, 1, 0, 3, 2, 3.

Simulating LRU in Software


R bits for pages 0-5, clock tick 0
1 0 1 0 1 1
R bits for pages 0-5, clock tick 1
1 1 0 0 1 0
R bits for pages 0-5, clock tick 2
1 1 0 1 0 1
R bits for pages 0-5, clock tick 3
1 0 0 0 1 0
R bits for pages 0-5, clock tick 4
0 1 1 0 0 0
Page
0
1
2
3
4
5
(a)
(b)
(c)
(d)
(e)

Page	(a)	(b)	(c)	(d)	(e)
0	10000000	11000000	11100000	11110000	01111000
1	00000000	10000000	11000000	01100000	10110000
2	10000000	01000000	00100000	00100000	10010000
3	00000000	00000000	10000000	01000000	00100000
4	10000000	11000000	01100000	10110000	01011000
5	10000000	01000000	10100000	01010000	00101000

Figure 3-18. 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 Page Replacement (1)

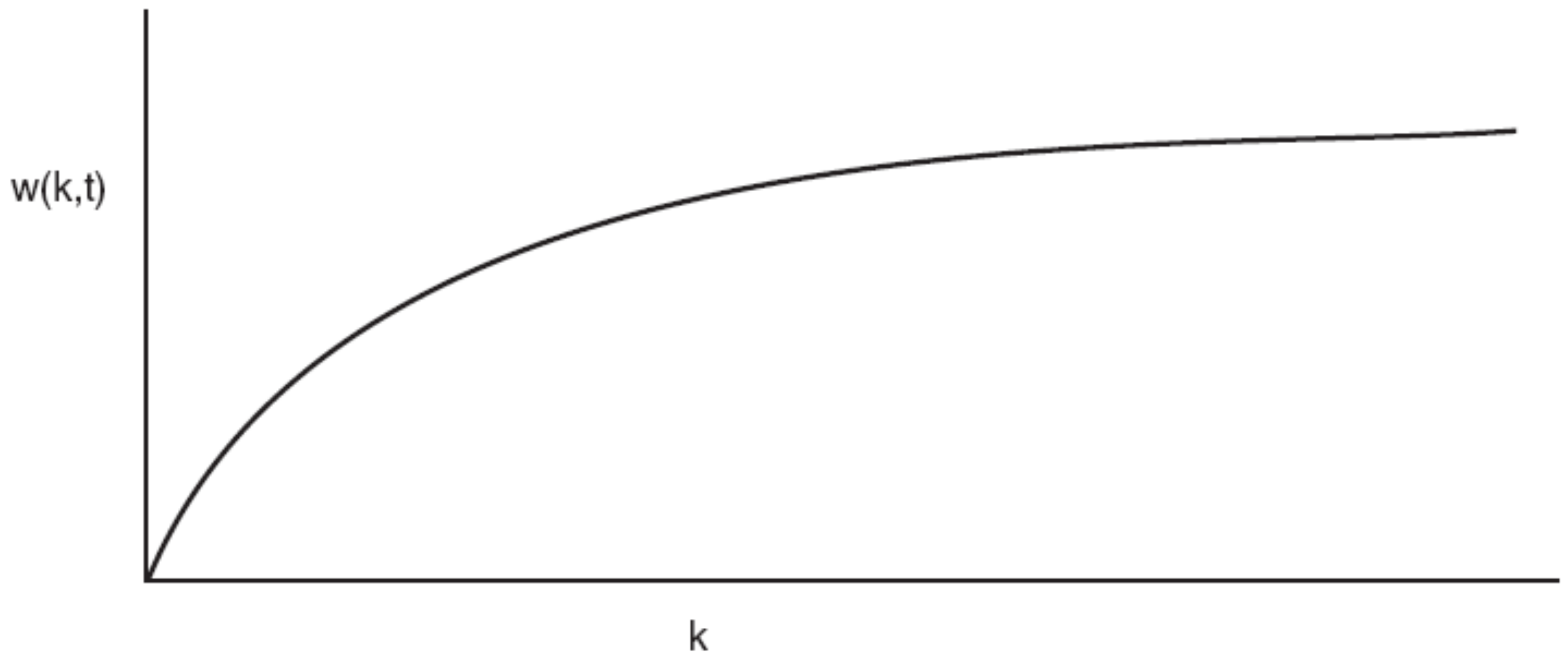
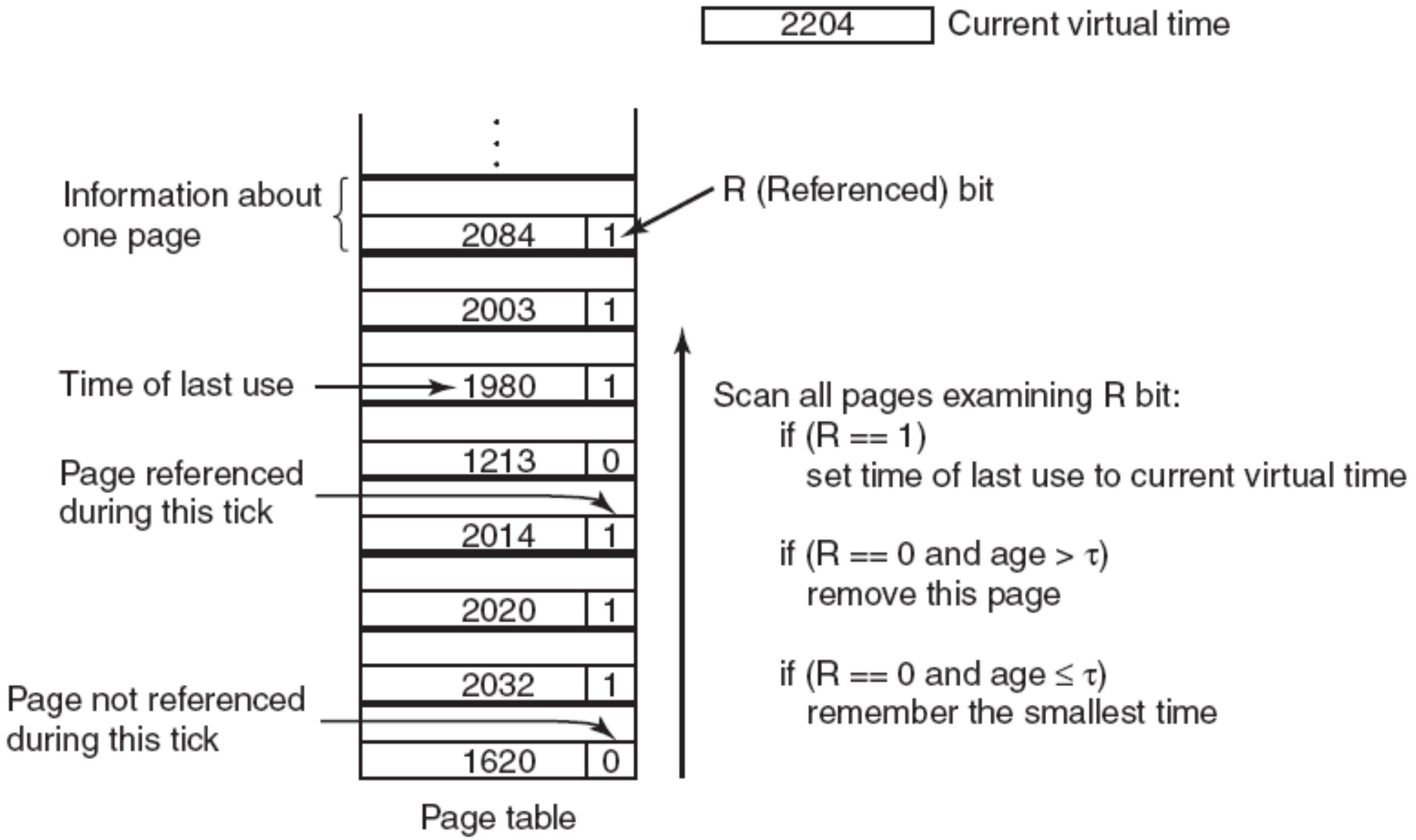


Figure 3-19. The working set is the set of pages used by the k most recent memory references. The function w(k, t) is the size of the working set at time t.

Working Set Page Replacement (2)



```
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
```

Figure 3-20. The working set algorithm.

The WSClock Page Replacement Algorithm (1)

When the hand comes all the way around to its starting point there are two cases to consider:

1. At least one write has been scheduled.
2. No writes have been scheduled.

The WSClock Page Replacement Algorithm (2)

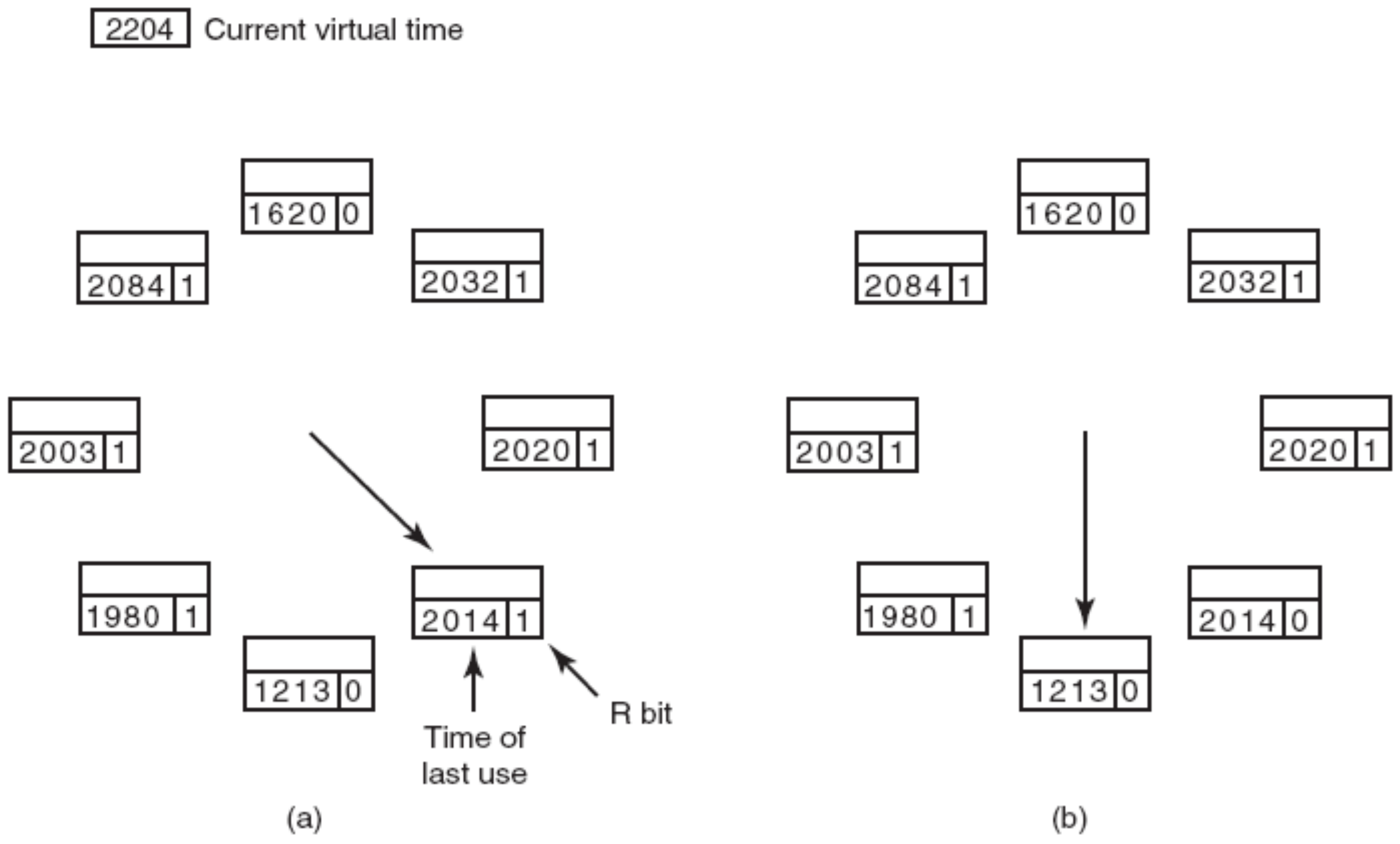


Figure 3-21. Operation of the WSClock algorithm. (a) and (b) give an example of what happens when R = 1.

The WSClock Page Replacement Algorithm (3)

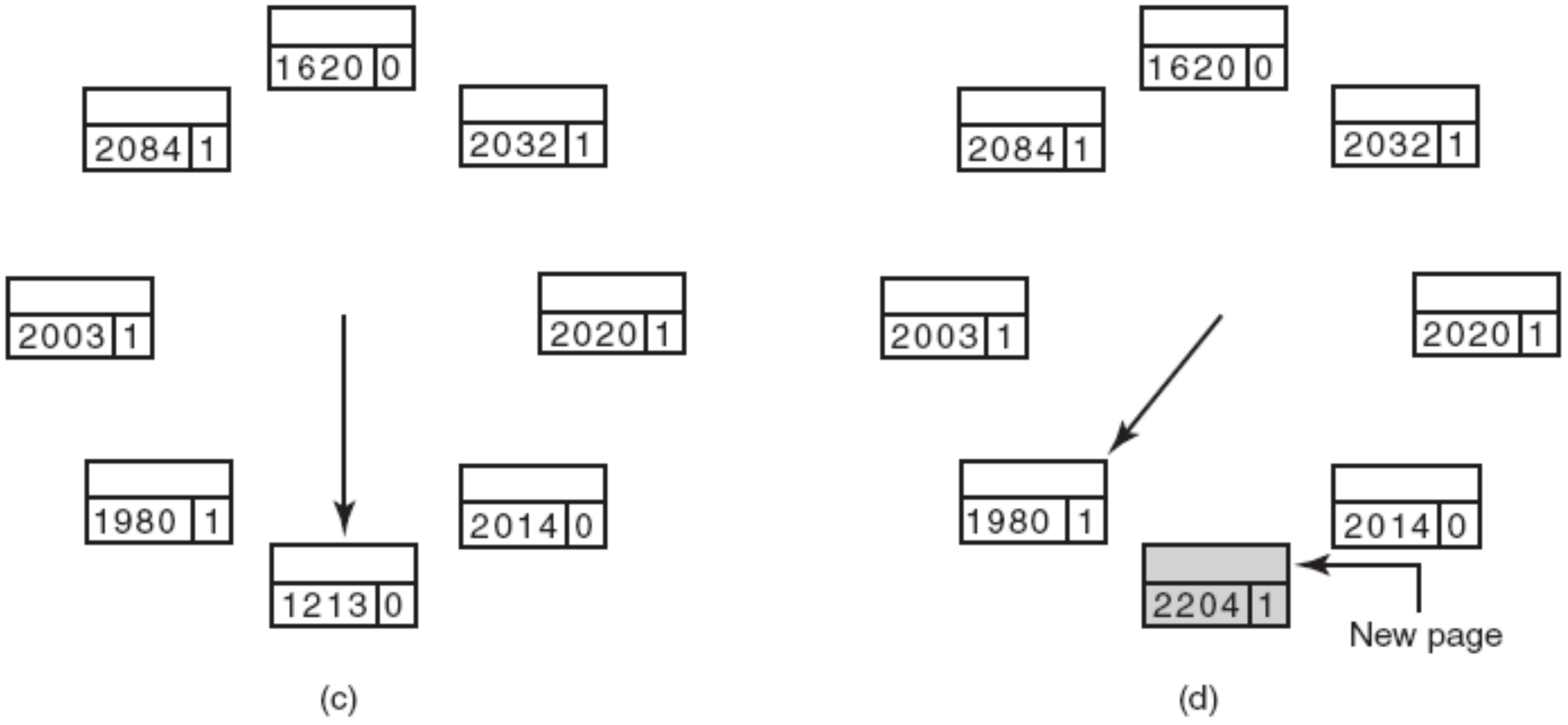
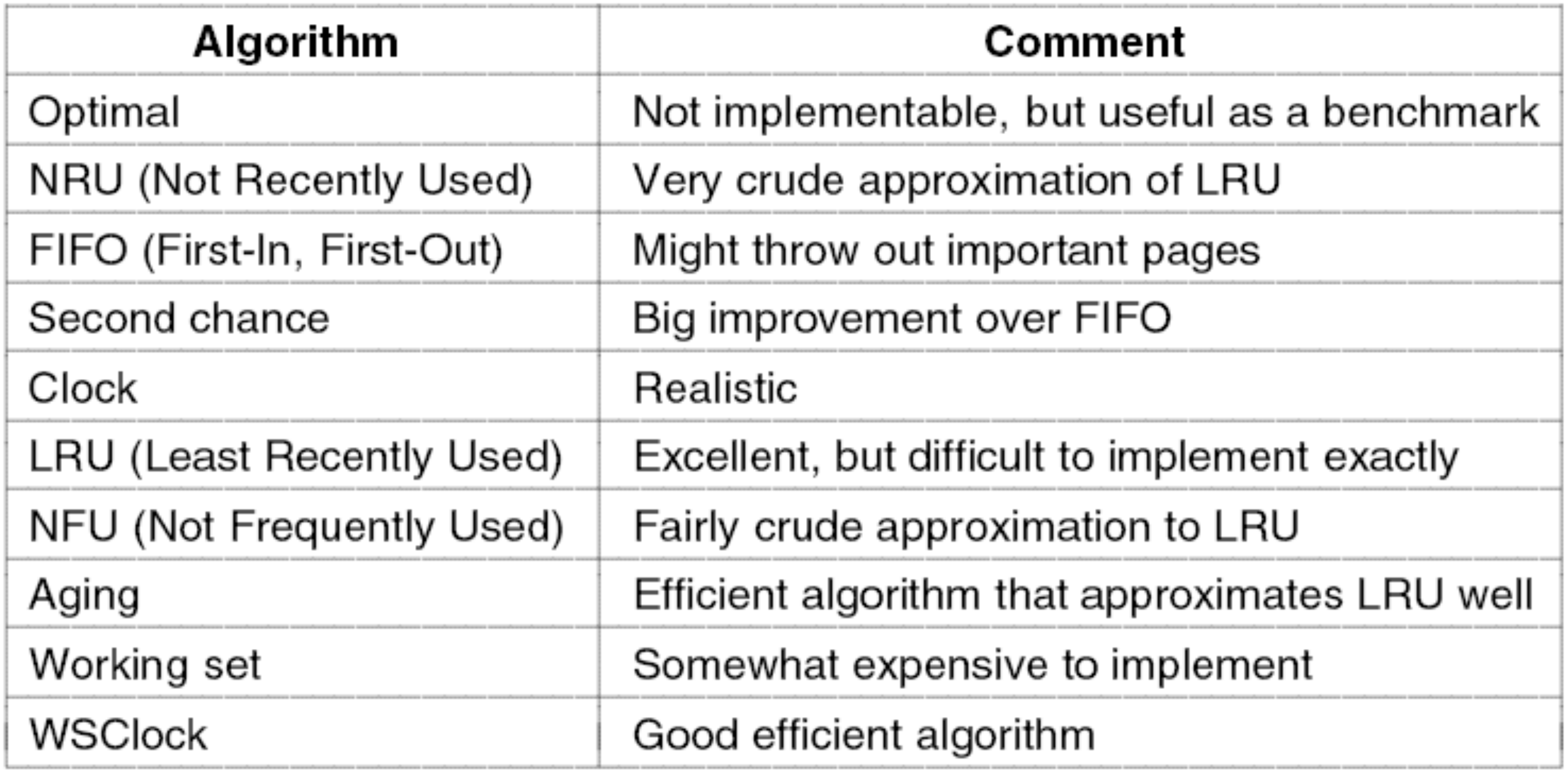


Figure 3-21. Operation of the WSClock algorithm. (c) and (d) give an example of R = 0.

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 3-22. Page replacement algorithms discussed in the text.

Local versus Global Allocation Policies (1)

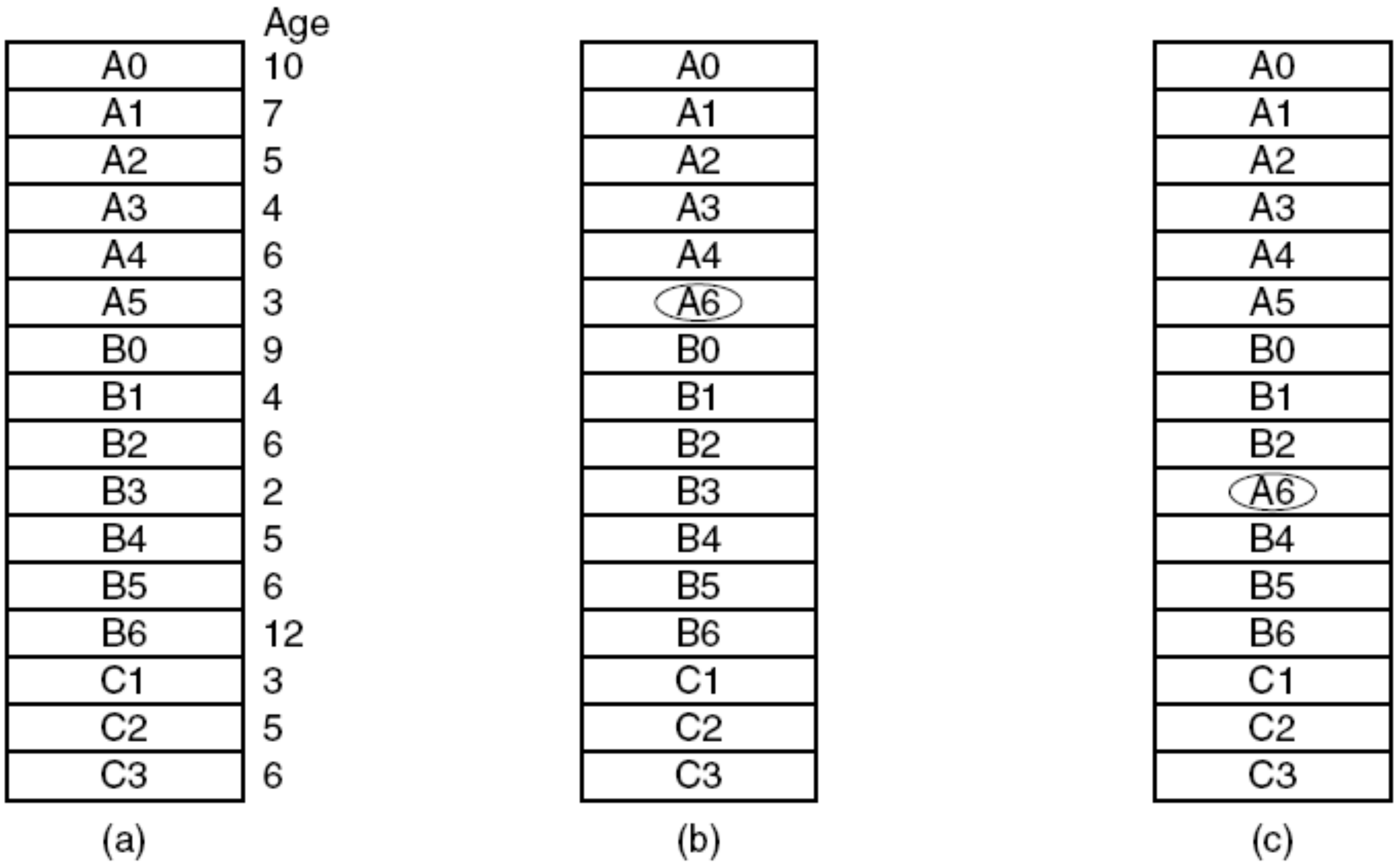

Age
A0 10
A1 7
A2 5
A3 4
A4 6
A5 3
B0 9
B1 4
B2 6
B3 2
B4 5
B5 6
B6 12
C1 3
C2 5
C3 6
(a)
A0 A1 A2 A3 A4 A6 B0 B1 B2 B3 B4 B5 B6 C1 C2 C3
(b)
A0 A1 A2 A3 A4 A5 B0 B1 B2 A6 B4 B5 B6 C1 C2 C3
(c)

Figure 3-23. Local versus global page replacement. (a) Original configuration. (b) Local page replacement. (c) Global page replacement.

Local versus Global Allocation Policies (2)

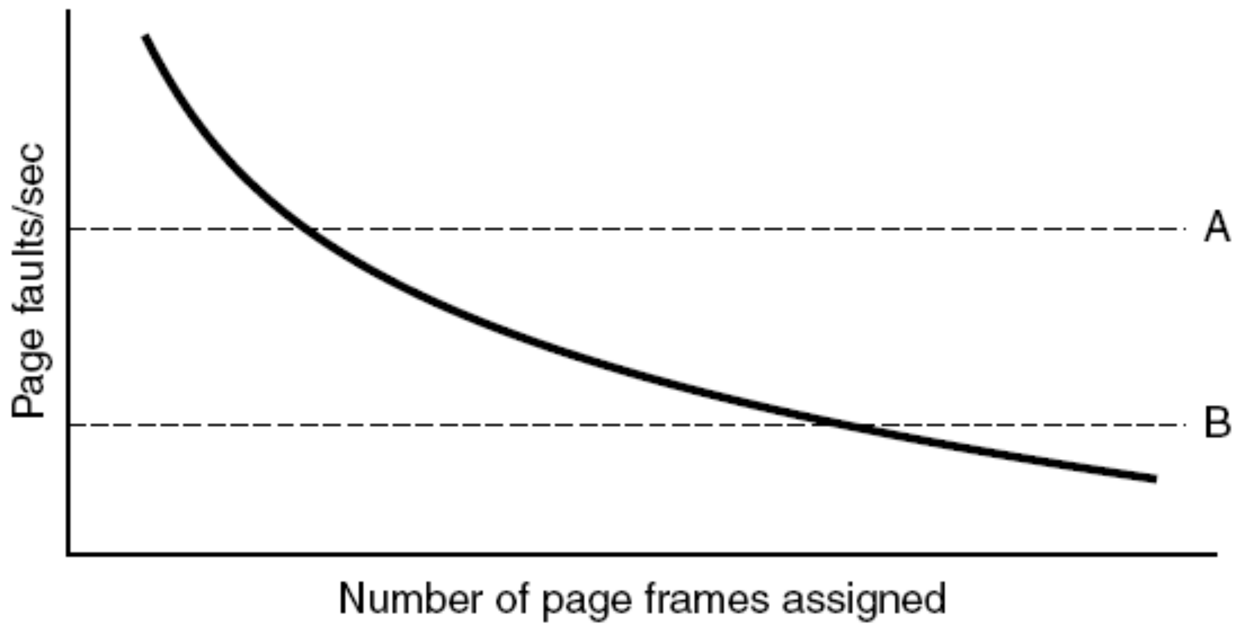


Figure 3-24. Page fault rate as a function of the number of page frames assigned.

Separate Instruction and Data Spaces

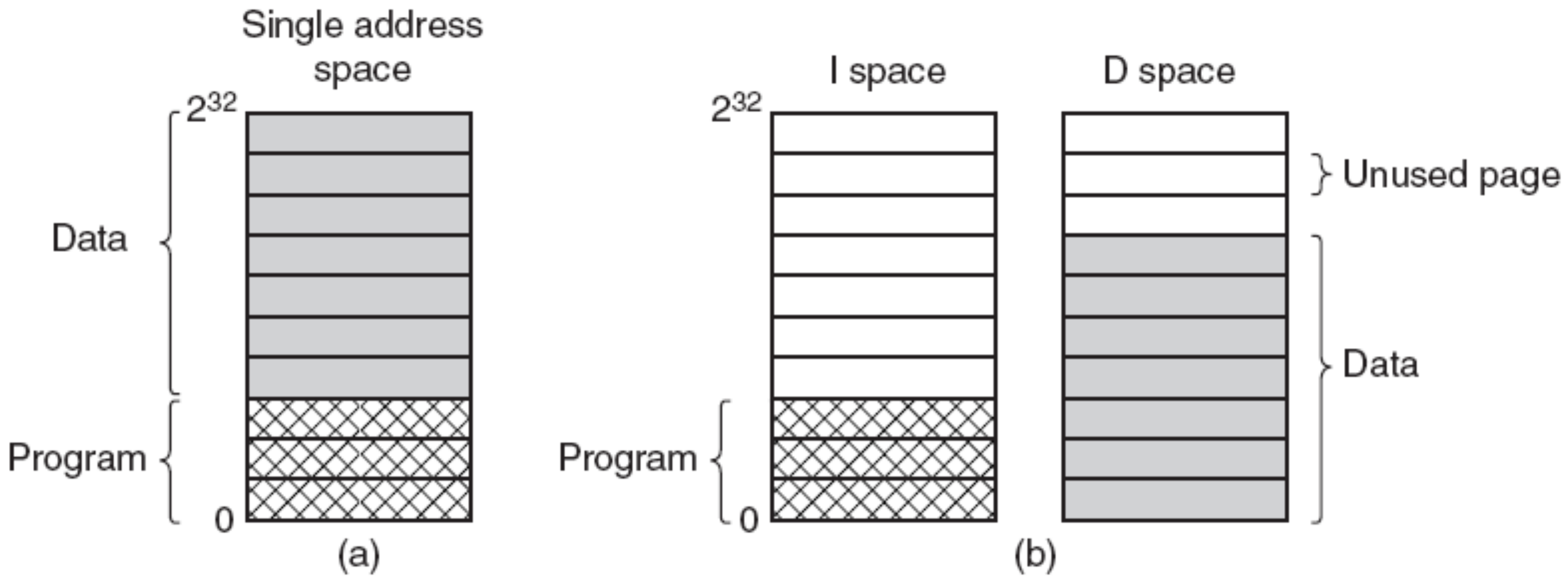


Figure 3-25. (a) One address space.
(b) Separate I and D spaces.

Shared Pages

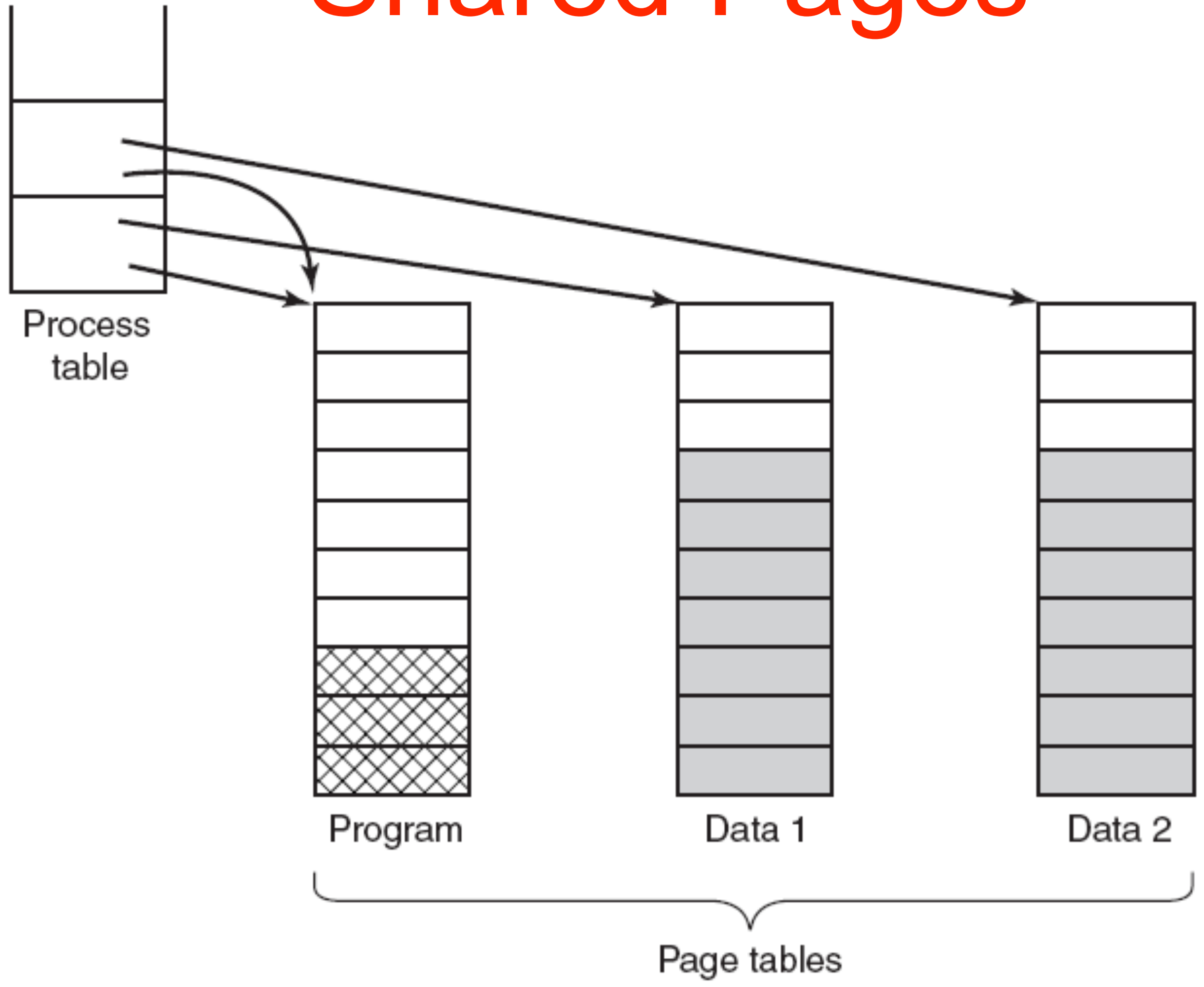


Figure 3-26. Two processes sharing the same program sharing its page table.

Shared Libraries

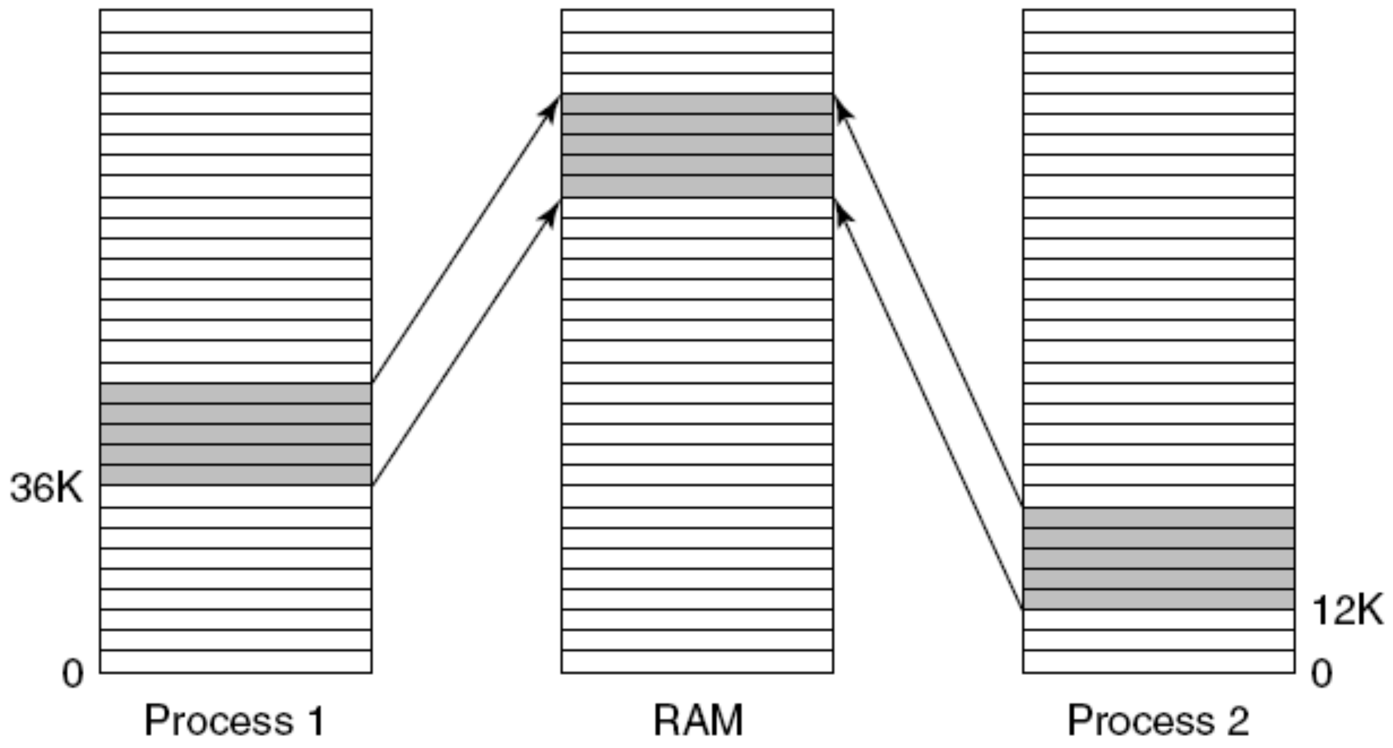


Figure 3-27. A shared library being used by two processes.

Page Fault Handling (1)

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

Page Fault Handling (2)

5. If the page frame selected is dirty, the page is scheduled for transfer to the disk, and a context switch takes place.
6. When page frame is clean, operating system looks up the disk address where the needed page is, schedules a disk operation to bring it in.
7. When disk interrupt indicates page has arrived, page tables updated to reflect position, 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 reset to point to that instruction.
9. Faulting process scheduled, operating system returns to the (assembly language) routine that called it.
10. This routine reloads registers and other state information and returns to user space to continue execution, as if no fault had occurred.

Instruction Backup

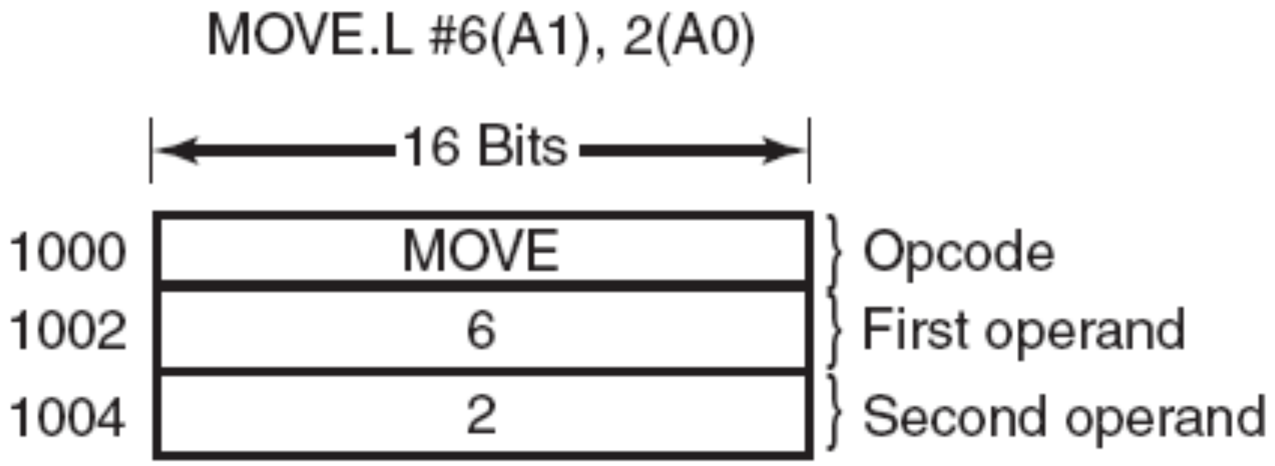


Figure 3-28. An instruction causing a page fault.

Backing Store (1)

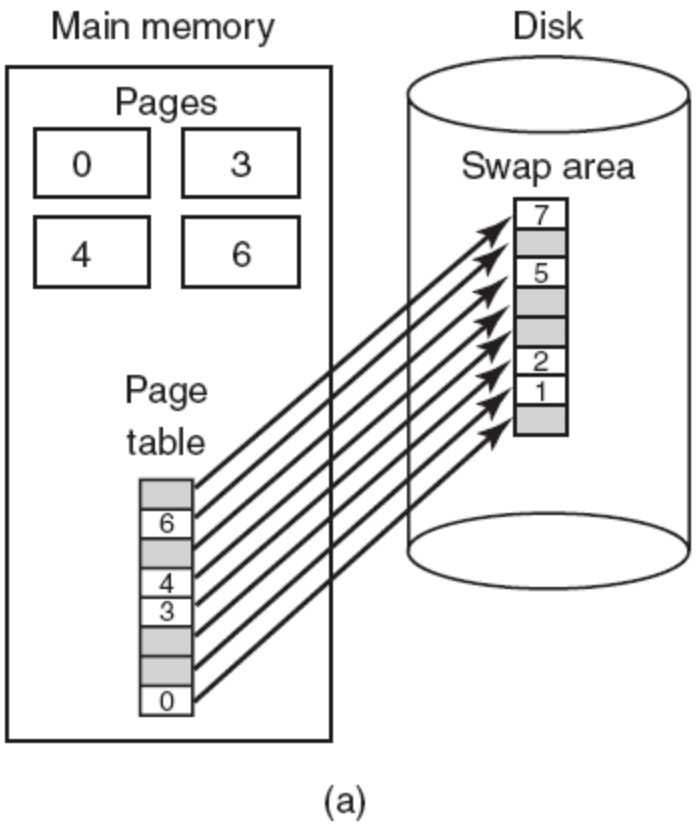


Figure 3-29. (a) Paging to a static swap area.

Backing Store (2)

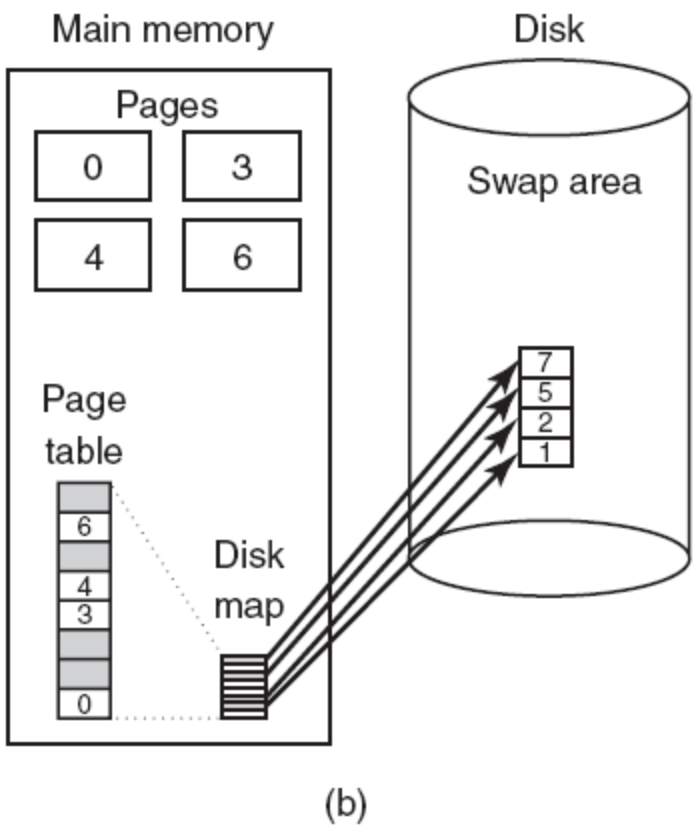


Figure 3-29. (b) Backing up pages dynamically.

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)

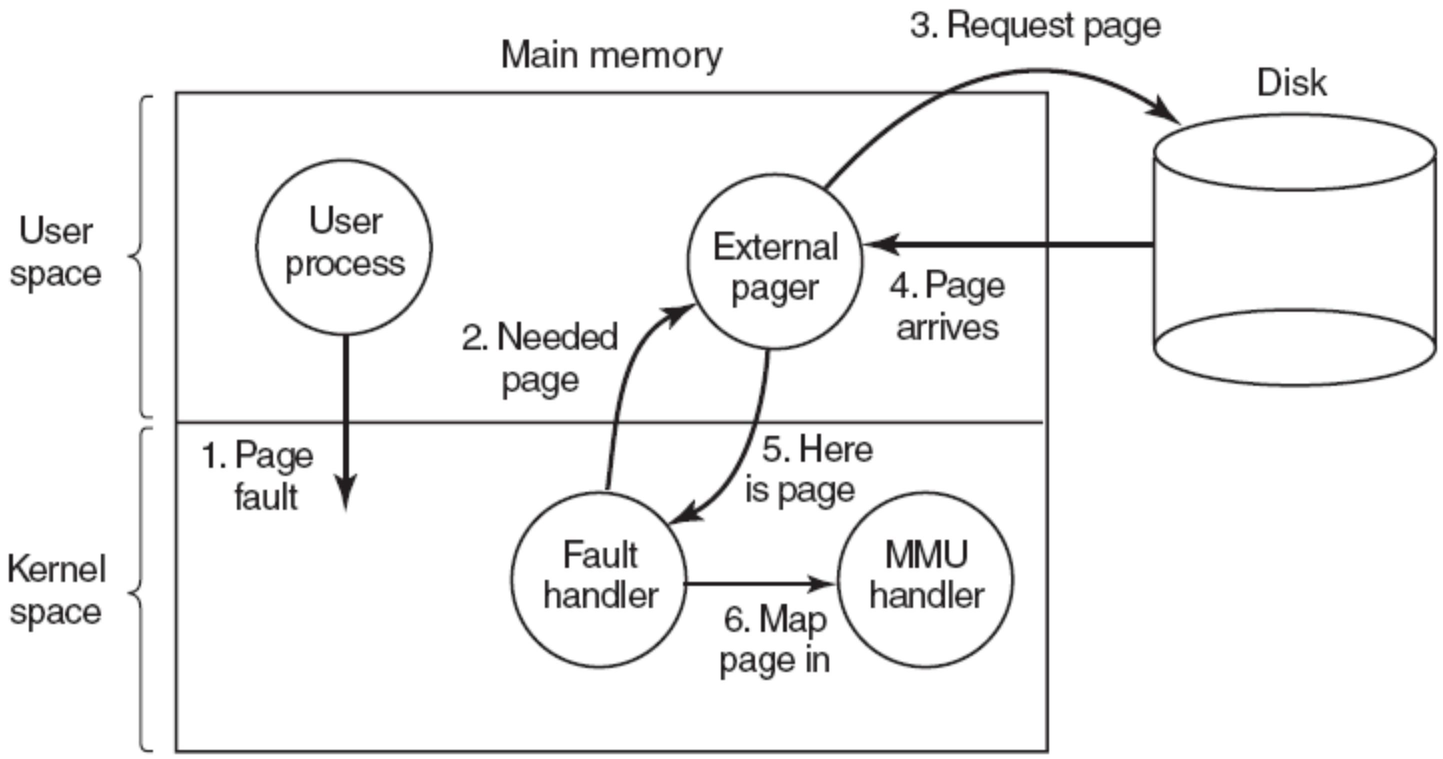


Figure 3-30. Page fault handling with an external pager.

Segmentation (1)

A compiler has many tables that are built up as compilation proceeds, possibly including:

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

Segmentation (2)

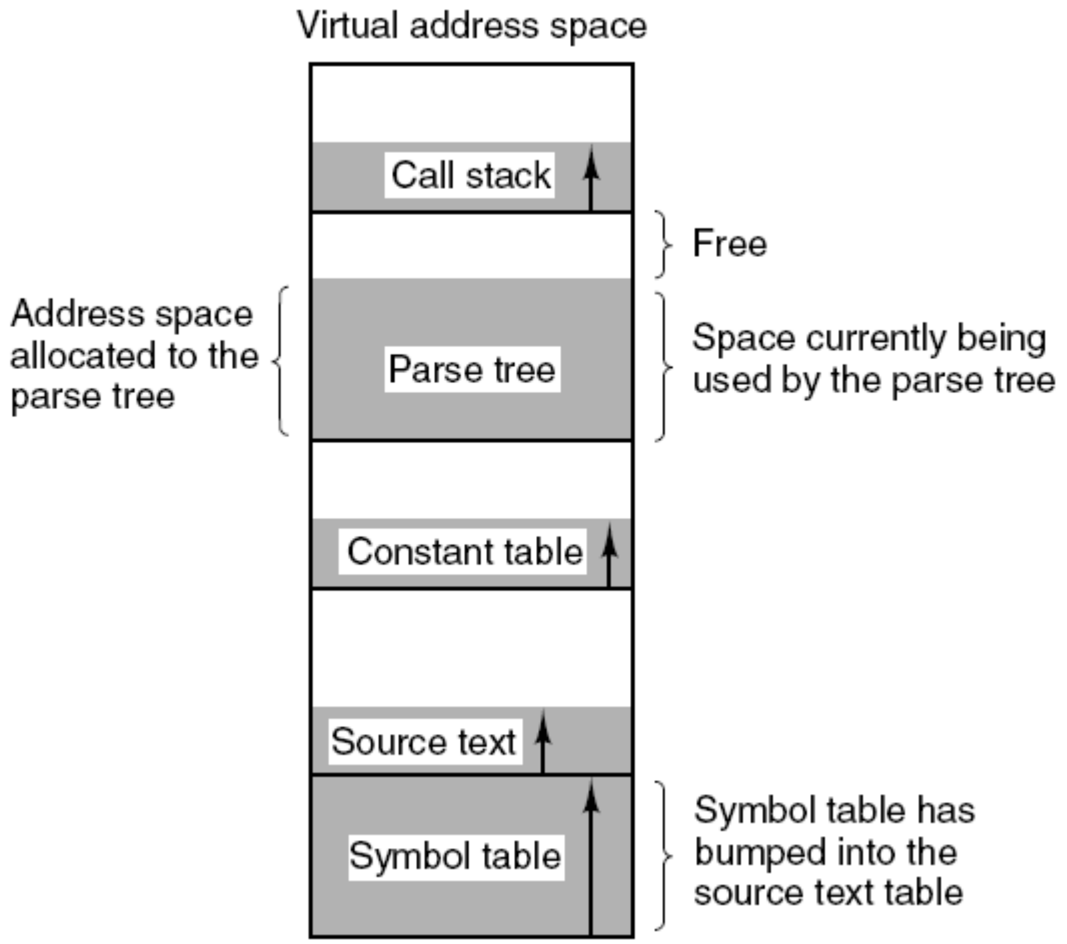

Virtual address space
Call stack
Free
Address space allocated to the parse tree
Parse tree
Space currently being used by the parse tree
Constant table
Source text
Symbol table
Symbol table has bumped into the source text table

Figure 3-31. In a one-dimensional address space with growing tables, one table may bump into another.

Segmentation (3)

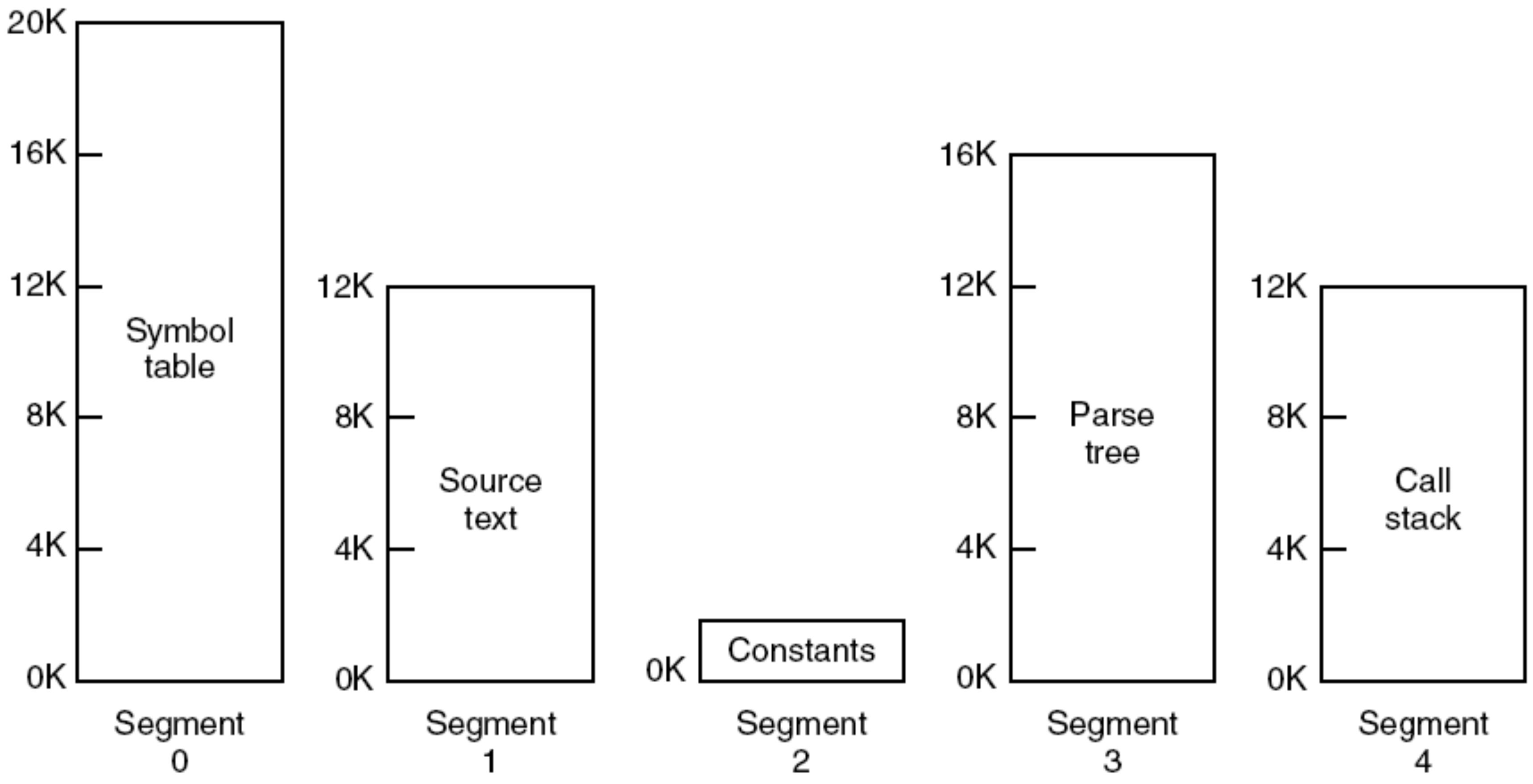


Figure 3-32. A segmented memory allows each table to grow or shrink independently of the other tables.

Implementation of Pure Segmentation

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 3-33. Comparison of paging and segmentation.

Segmentation with Paging: MULTICS (1)

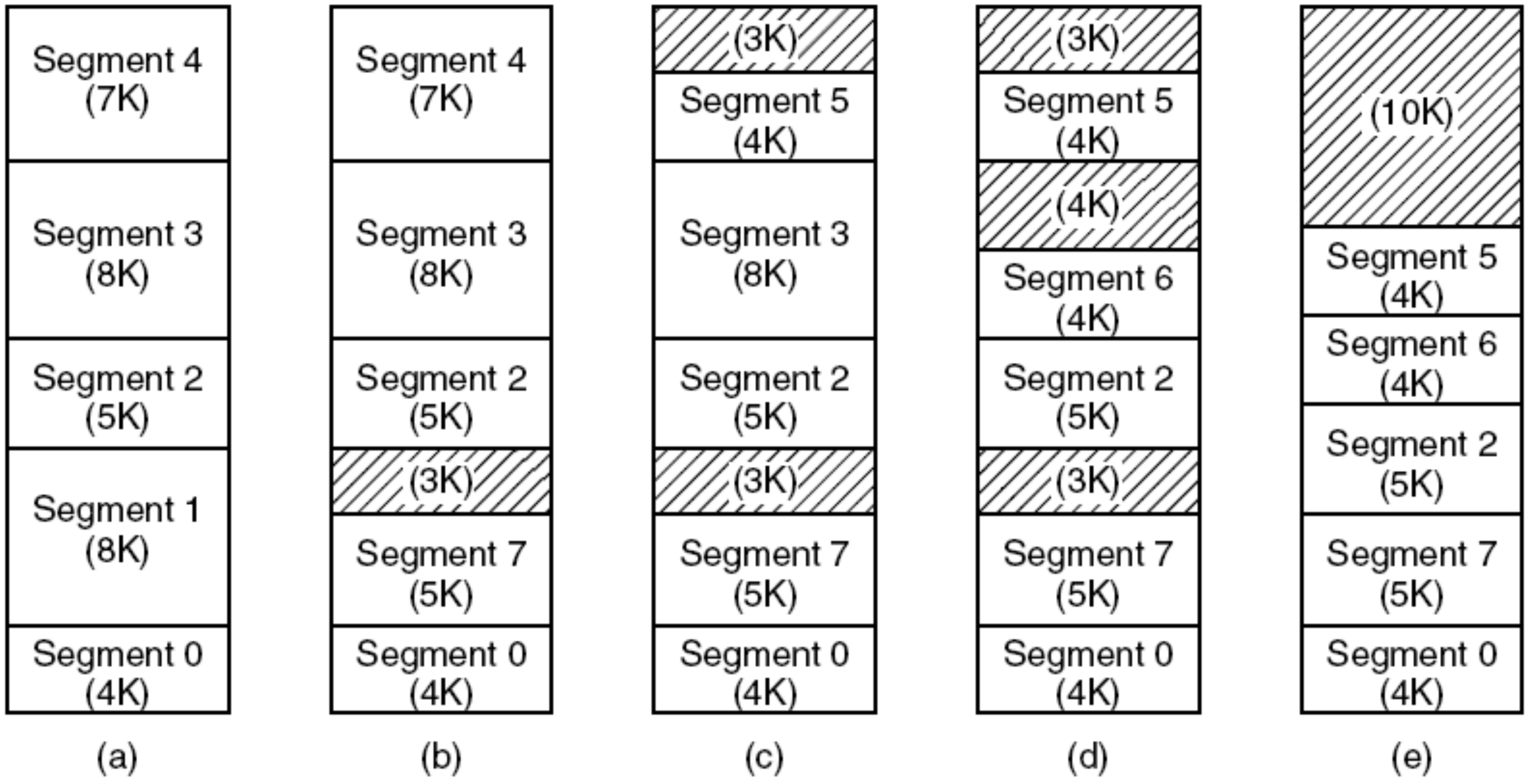
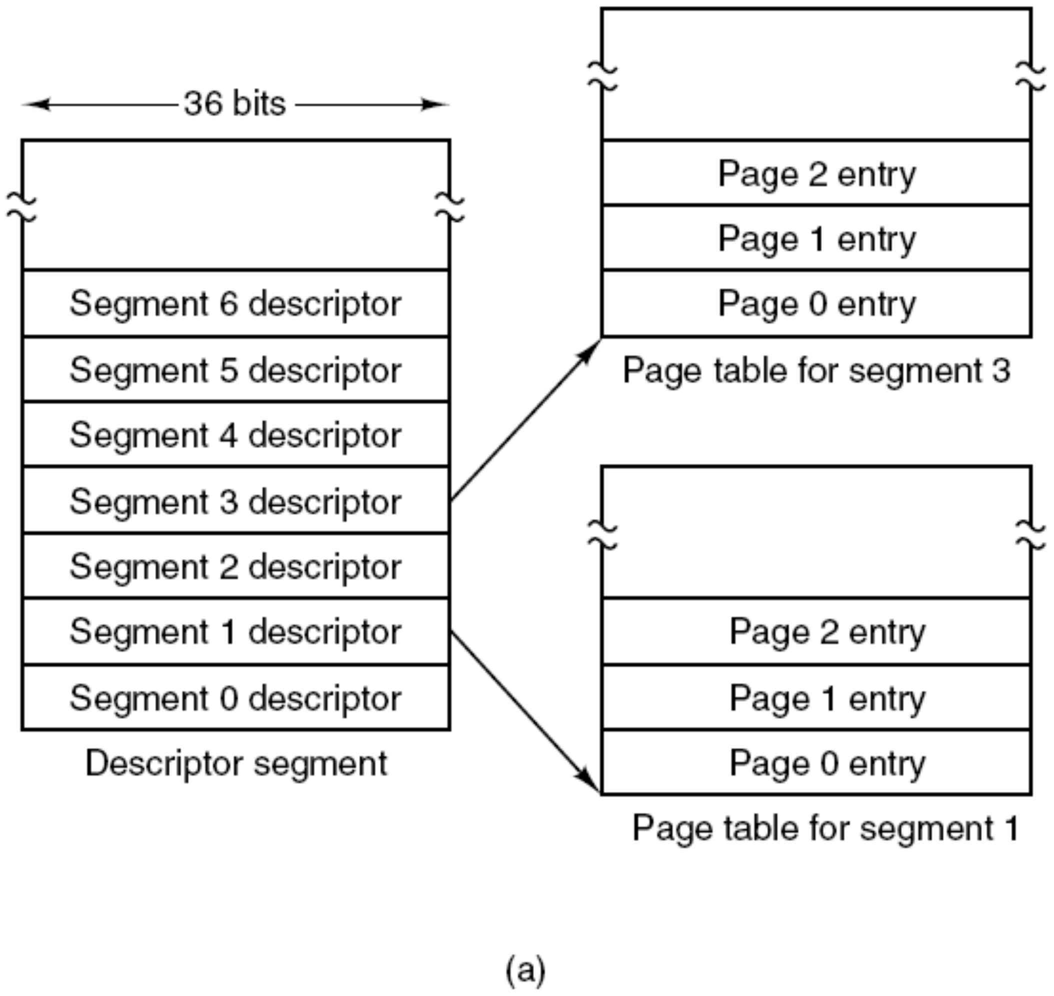


Figure 3-34. (a)-(d) Development of checkerboarding. (e) Removal of the checkerboarding by compaction.

Segmentation with Paging: MULTICS (2)

Figure 3-35. The MULTICS virtual memory. (a) The descriptor segment points to the page tables.



Segmentation with Paging: MULTICS (5)

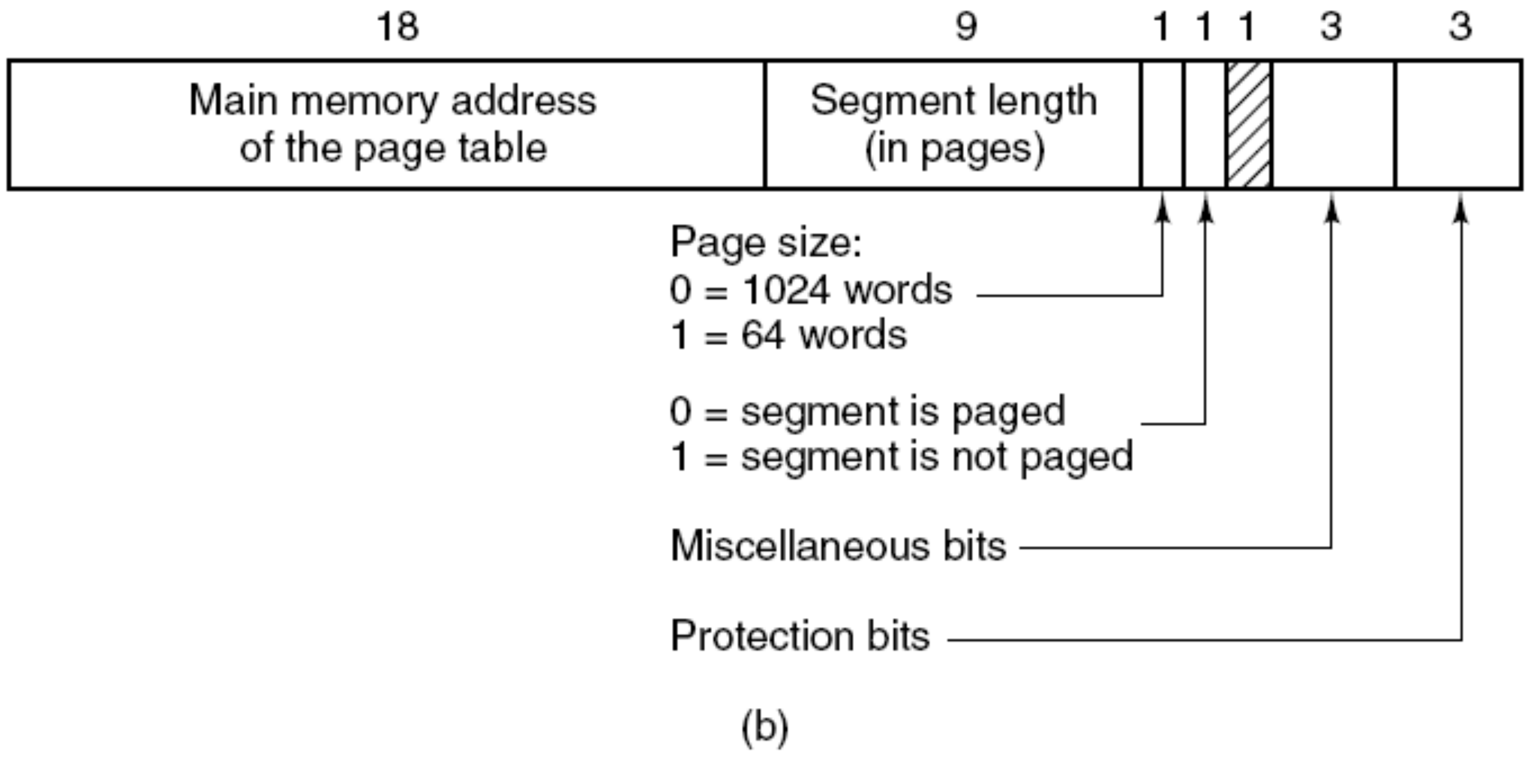


Figure 3-35. The MULTICS virtual memory. (b) A segment descriptor. The numbers are the field lengths.

Segmentation with Paging: MULTICS (6)

When a memory reference occurs, the following algorithm is carried out.

1. The segment number used to find segment descriptor.
2. Check is made to see if the segment's page table is in memory.
   a) If not, segment fault occurs.
   b) If there is a protection violation, a fault (trap) occurs.

Segmentation with Paging: MULTICS (7)

3. Page table entry for the requested virtual page examined.
   a) If the page itself is not in memory, a page fault is triggered.
   b) If it is in memory, the main memory address of the start of the page is extracted from the page table entry
4. The offset is added to the page origin to give the main memory address where the word is located.
5. The read or store finally takes place.

Segmentation with Paging: MULTICS (8)

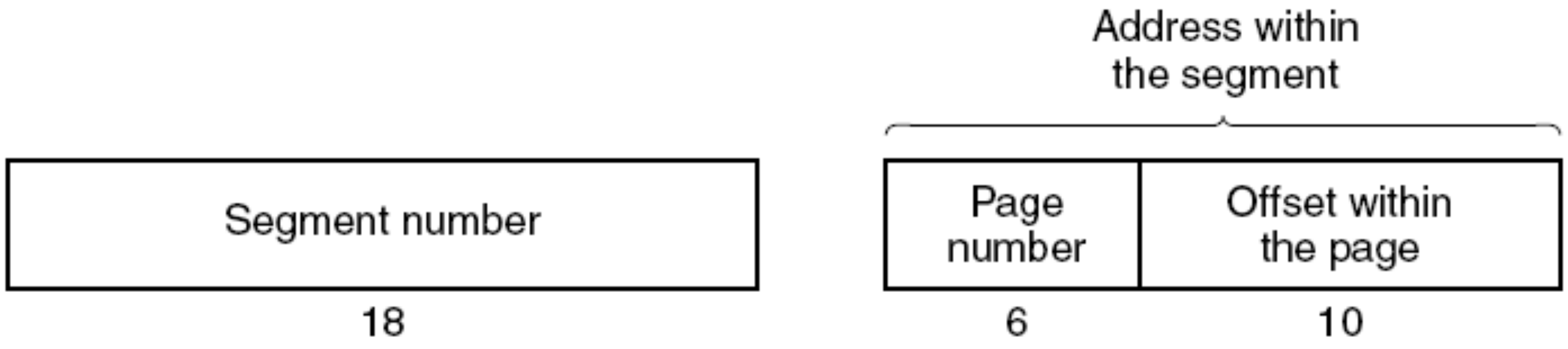


Figure 3-36. A 34-bit MULTICS virtual address.

Segmentation with Paging: MULTICS (9)

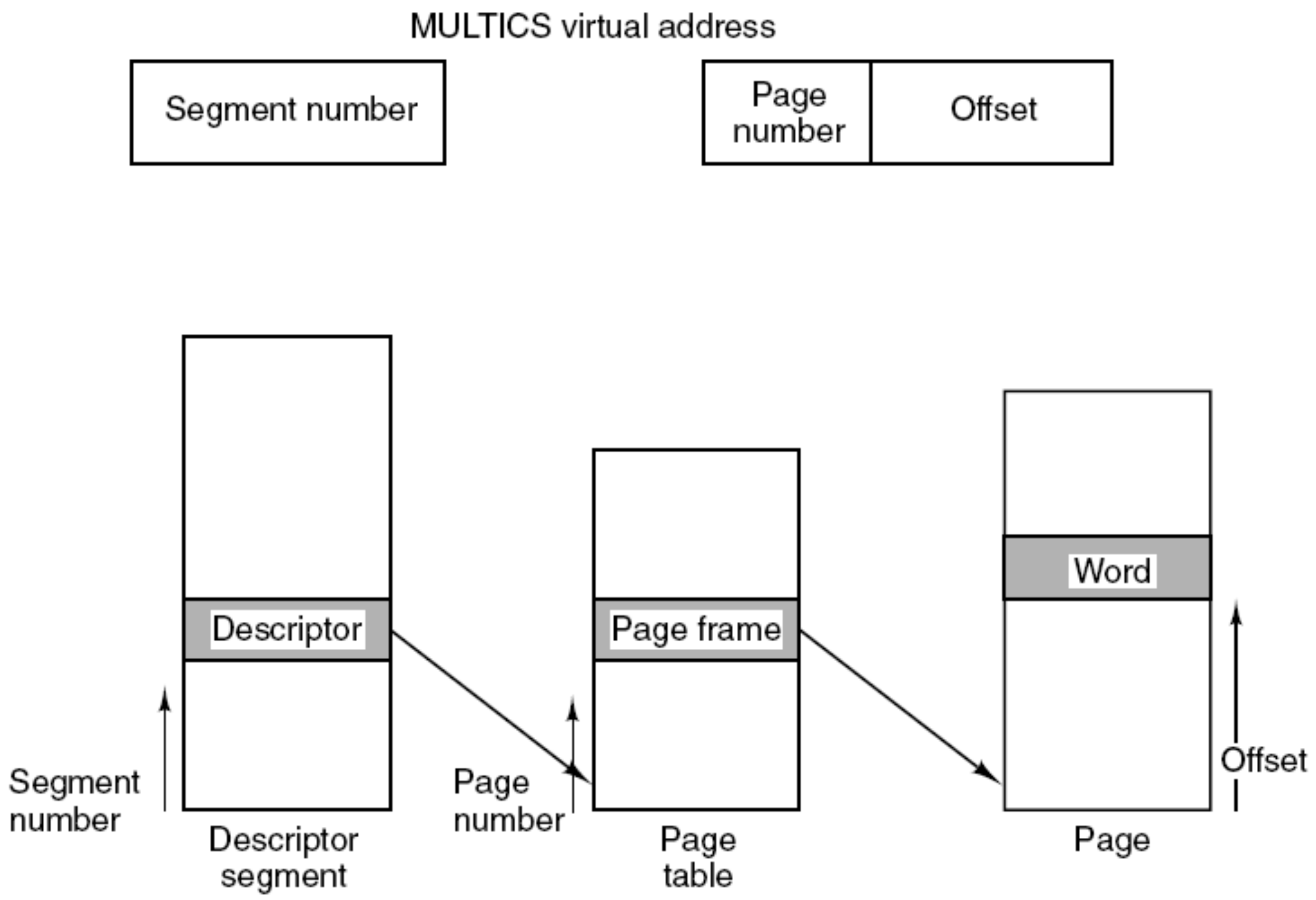
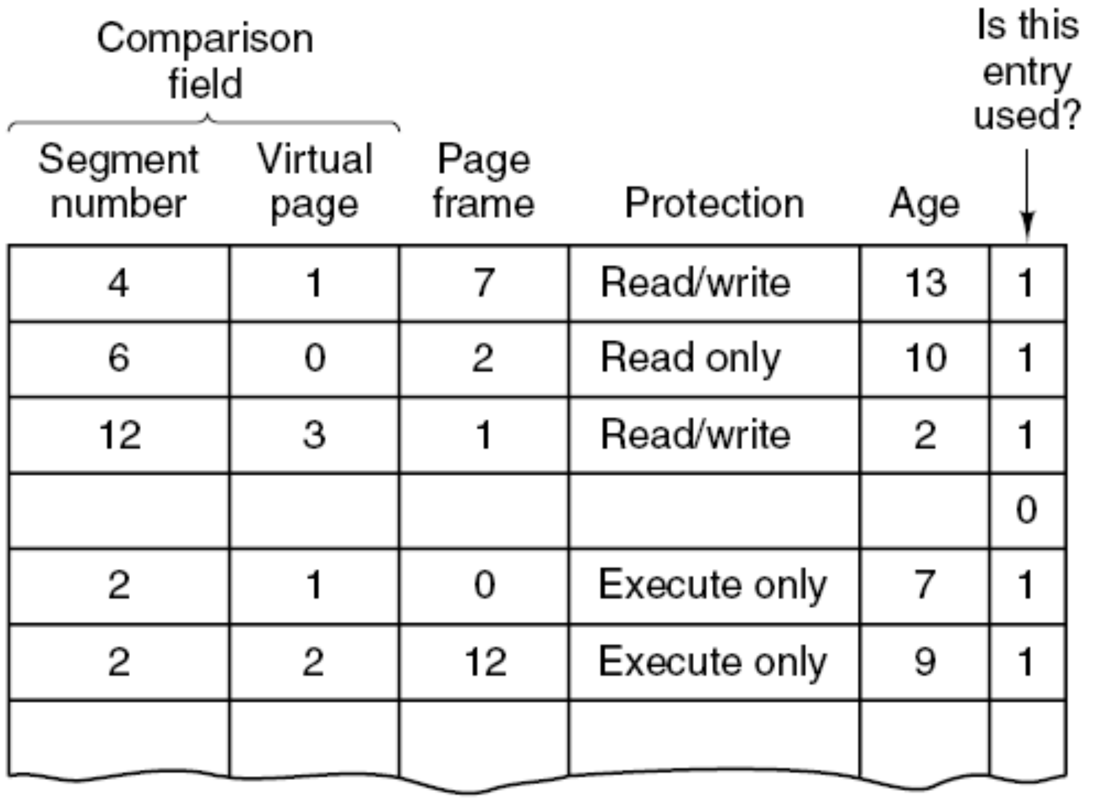


Figure 3-37. Conversion of a two-part MULTICS address into a main memory address.

Segmentation with Paging: MULTICS (10)


Comparison field
Is this entry used?

Segment number	Virtual page	Page frame	Protection	Age	
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 3-38. A simplified version of the MULTICS TLB. The existence of two page sizes makes the actual TLB more complicated.

Segmentation with Paging: The Intel Pentium (1)

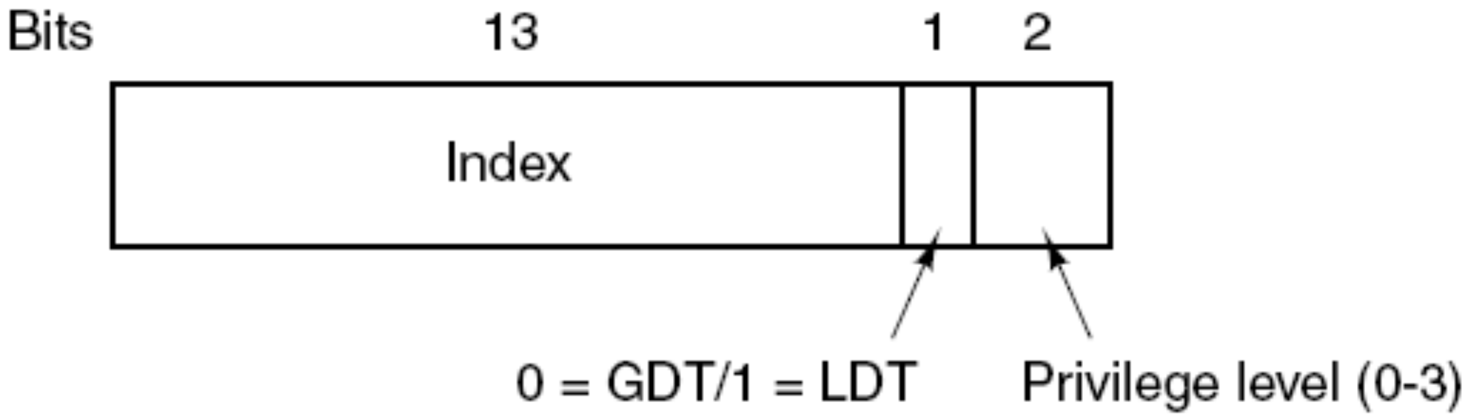


Figure 3-39. A Pentium selector.

Segmentation with Paging: The Intel Pentium (2)

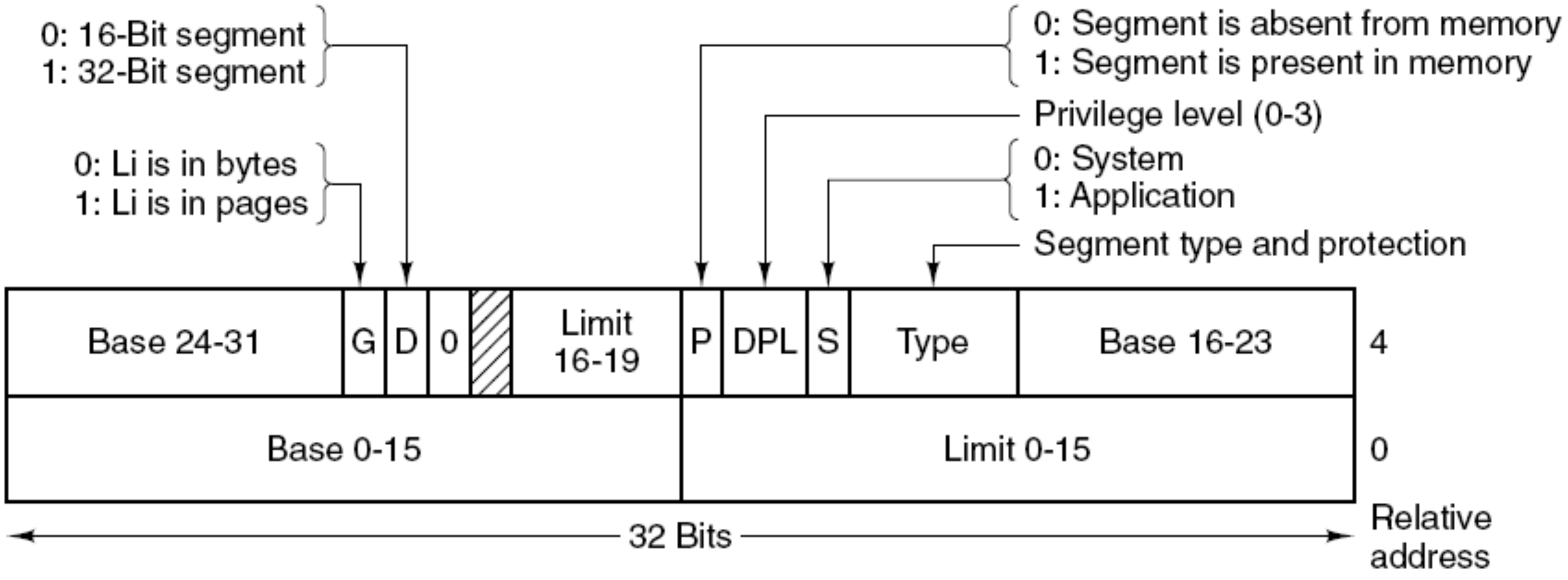


Figure 3-40. Pentium code segment descriptor. Data segments differ slightly.

Segmentation with Paging: The Intel Pentium (3)

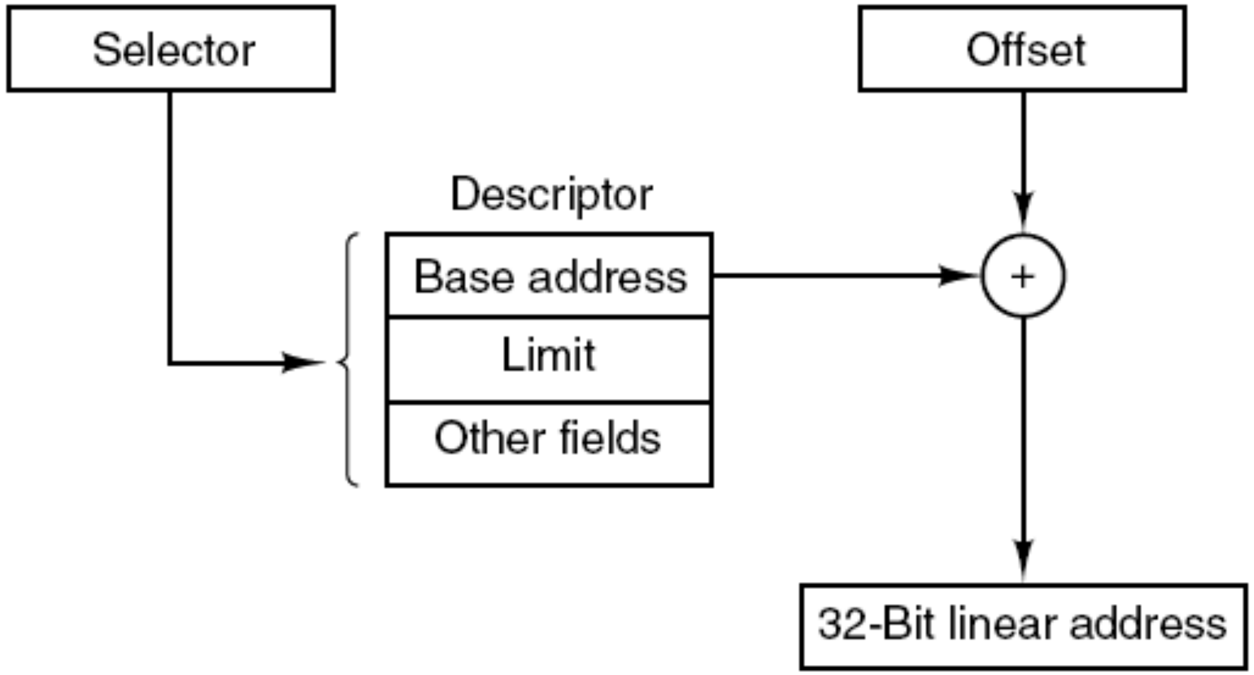


Figure 3-41. Conversion of a (selector, offset) pair to a linear address.

Segmentation with Paging: The Intel Pentium (4)

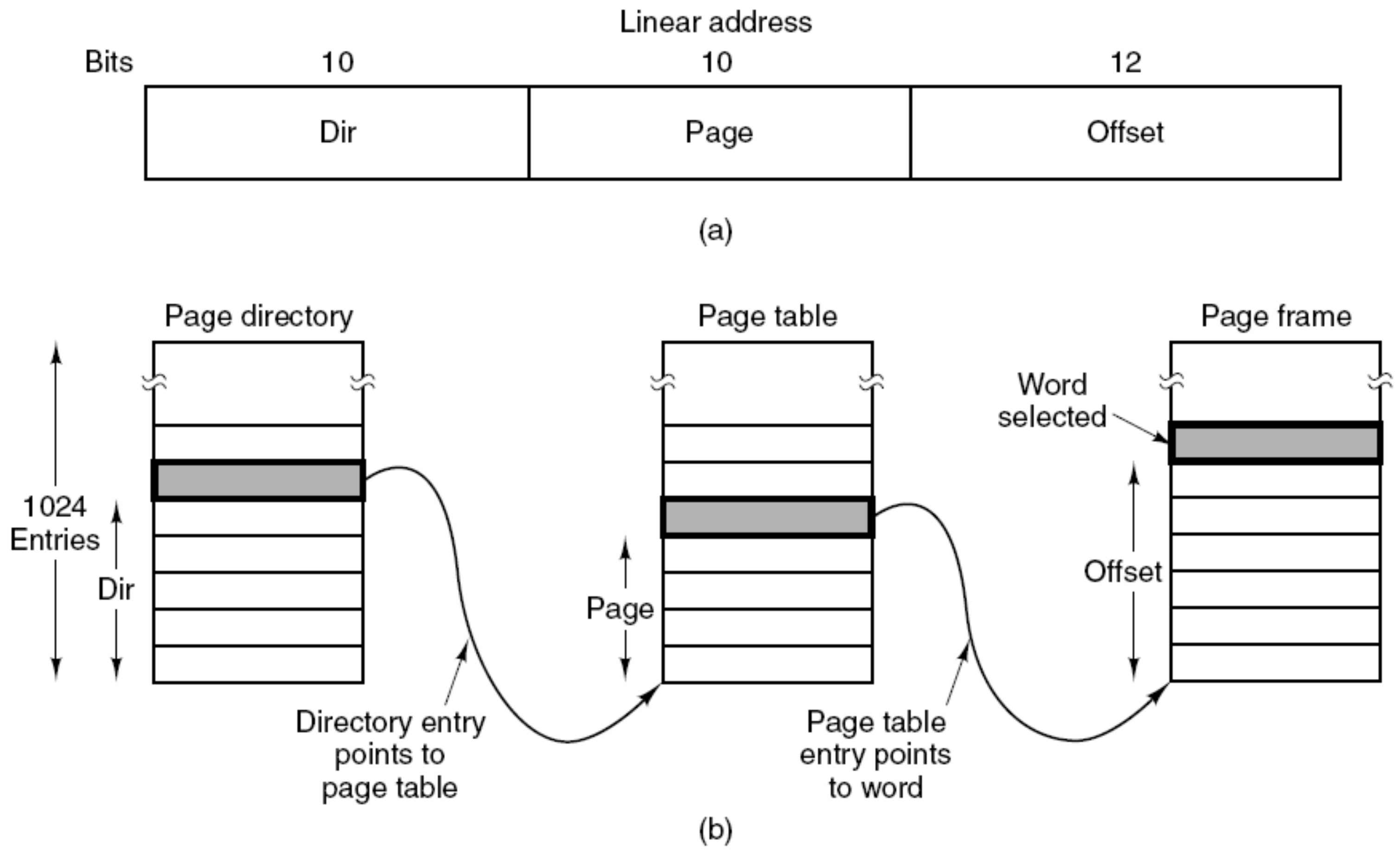


Figure 3-42. Mapping of a linear address onto a physical address.

Segmentation with Paging: The Intel Pentium (5)

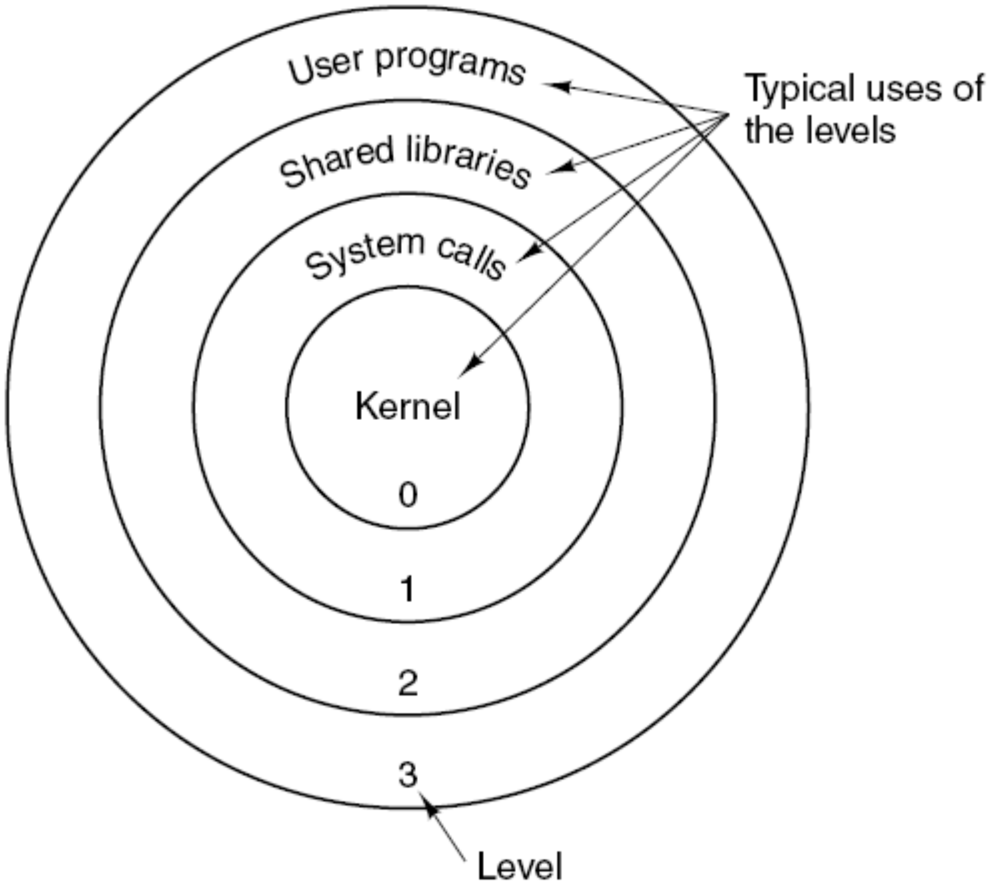


Figure 3-43. Protection on the Pentium.