现 代 操 作 系 统

Modern Operating Systems

中南大学计算机学院

Chapter 3 Memory Management

- 3.1 No Memory Abstraction
- 3.2 Address Spaces
- 3.3 Virtual Memory
- 3.4 Page replacement Algorithms
- 3.5 Design Issues for Paging Systems
- 3.6 Implementation Issues
- 3.7 Segmentation
- 3.8 Reading Materials

3.1 No Memory Abstraction

- **Every program simply access the physical memory as**
  - MOV REGISTER1, 1000
- **No possible to have two programs in memory at the same time.**
- **Three simple schemes of memory management**
  - OS be in RAM (at the bottom of memory)
  - OS be in ROM (at the top of memory)
  - OS be in RAM and Device drivers be in ROM(BIOS)

3.1 No Memory Abstraction

- **Running multiple programs without a memory abstraction**
  - Swapping
  - relocation

3.2 Address Spaces

- **Several major drawbacks**
  - User programs can easily trash the OS intentionally or by accident
  - Difficult to have multiple programs running at once
- **Notion of Address Space**
  - Solve two problems: protection and relocation
  - Address Space: the set of addresses that a process can use to address memory.
  - Dynamic relocation——base register & limit register
  - Swapping
  - Memory compaction
  - Managing free memory——Bitmap & linked list
  - Allocation algorithms——first fit & next fit & best fit & worst fit & quick fit

习题

1、紧凑：假设有很多空闲区和数据段随机分布，读或写32位长度字需要10ns的时间，紧凑128MB大概需要多长时间？

2、在一个交换系统中，按内存地址排列的空闲区大小是：10KB、4KB、20KB、18KB、7KB、9KB、12KB和15KB。对于连续的段请求：1）12KB；2）10KB；3）9KB，使用first-fit，将找出哪个空闲区？使用best-fit、worst-fit以及next-fit呢？

3.3 Virtual Memory

- **Overlay——adopted in the 1960s**
- **The main idea of Virtual Memory**
  - Each program has its own address space, which is broken up into chunks (pages).
  - Each page is a contiguous range of addresses.
  - All pages are mapped onto physical memory, but not all pages have to be in physical memory to run the program.
  - When the program references a part of its address space that is in physical memory, the hardware performs the necessary mapping on the fly. If not in physical memory , the OS get the missing piece firstly and re-execute the instruction that failed.

3.3 Virtual Memory

- **Paging——Demand Paging**
  - Virtual address——virtual address space
  - Memory Management Unit——MMU
  - Page table: virtual page number, page frame number, offset, Present/absent bit, protection bit, modified bit, referenced bit, caching disabled
  - Page fault
  - Speeding up paging——TLB in MMU——soft miss & hard miss
  - Page tables for large memories——multilevel page table & inverted page table
- **Segmentation——Demand Segmentation**

Paging

- Hardware support for paging
  - page table v/i bit
  - swap space (hard disk)
- Saving Swapped pages
  - If page has not been modified, just overwrite.
  - If page has been modified, write to swap space.
- **Locality of Reference**
  - ***Spatial locality***
  - ***Temporal locality***

Paging Performance

- Page fault rate 0 <= p <= 1
  - 0 = no page faults
  - 1 = always page fault
- **Effective Access Time**
  - EAT = (1 - p) * memory access +
    p (page fault overhead +
    [swap page out] +swap page in +
    restart overhead)

Performance Examples

- Parameters:
  - memory access time = 20 ns = 0.02 μsec
  - 50% of pages swapped have been modified
  - Page Swap time: 10 msec = 10,000 μsec
  - p = **0.01**; 0.000001
- **Performance**

  EAT = .99 * 0.02 + 0.01 * 15,000 = 150.0198 μsec

Performance Examples

- Parameters:
  - memory access time = 20 ns = 0.02 μsec
  - 50% of pages swapped have been modified
  - Page Swap time: 10 msec = 10,000 μsec
  - p = 0.01; **0.000001**
- **Performance**

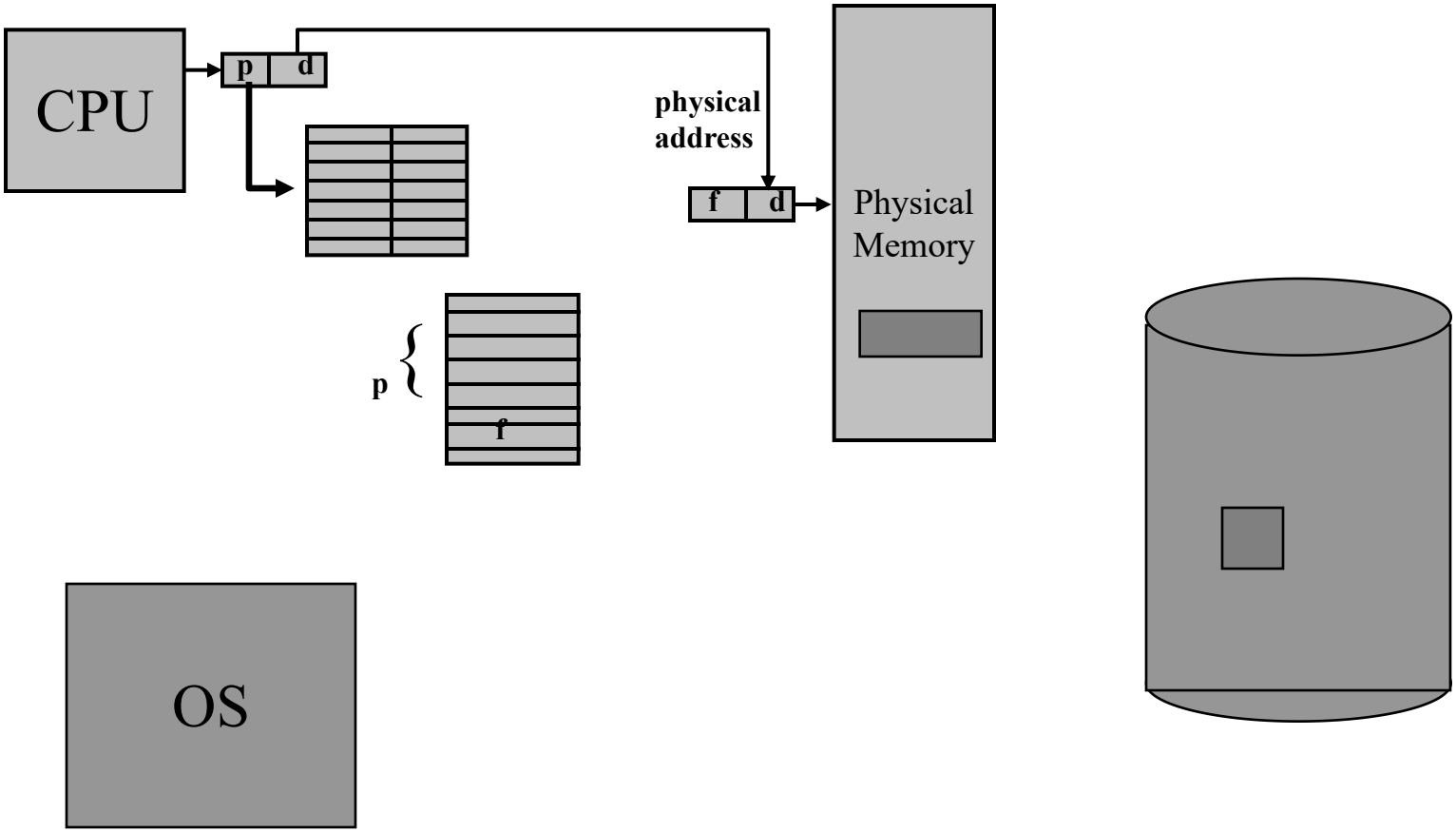
  EAT = .99 * 0.02 + 0.01 * 15,000 = 150.0198 μsec

  EAT = .999999 * 0.02 + 0.000001 * 15,000 = .015 + .0199 = 0.035 μsec = 35 ns

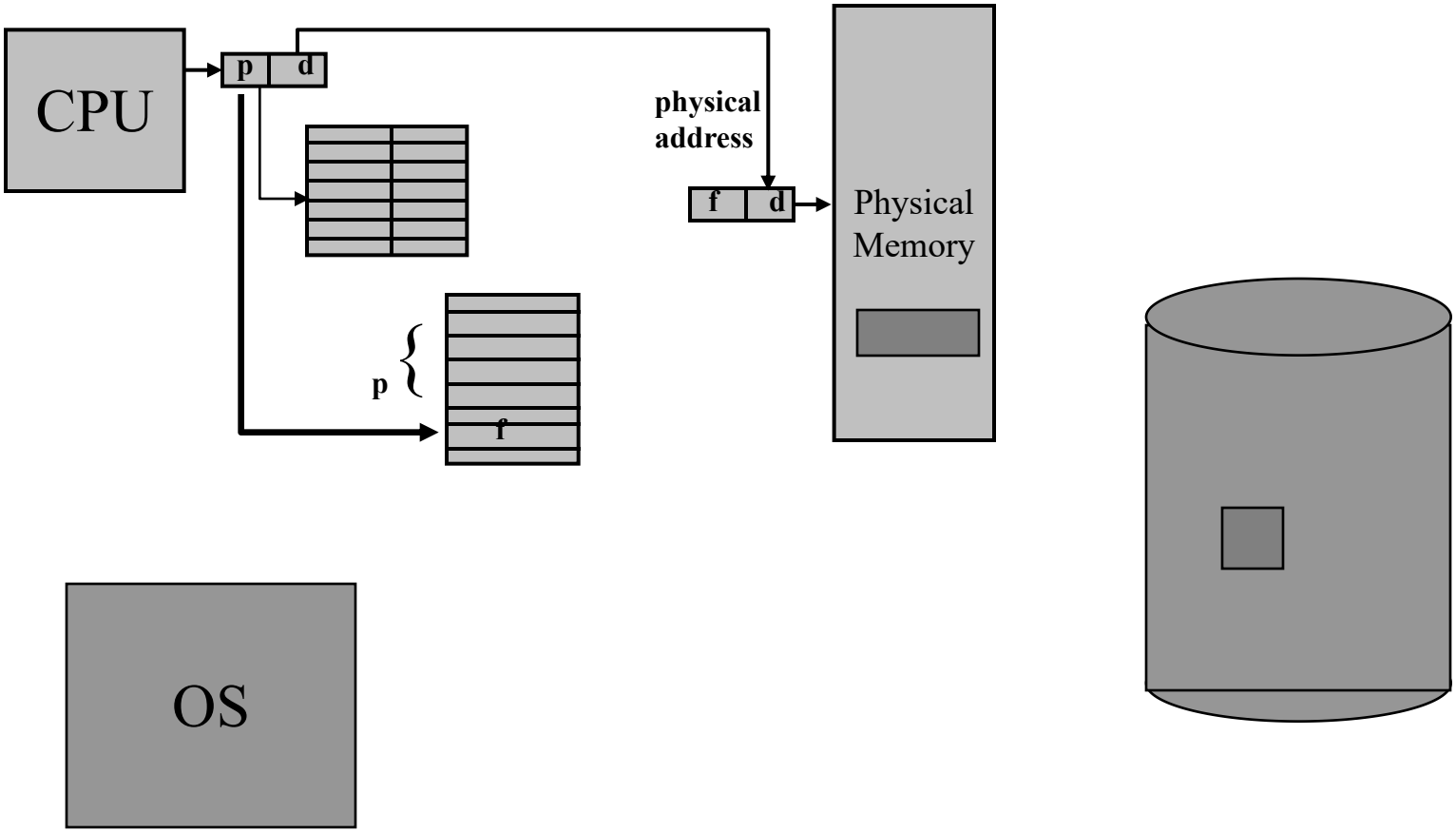
Page Replacement

- Find the desired page on the disk
- Find a free frame
  - If there is a free frame, use it
  - If not, use some algorithm to find a "victim"
  - Write the victim page to disk and update page table
- Read in desired page from disk
- **Restart user process**

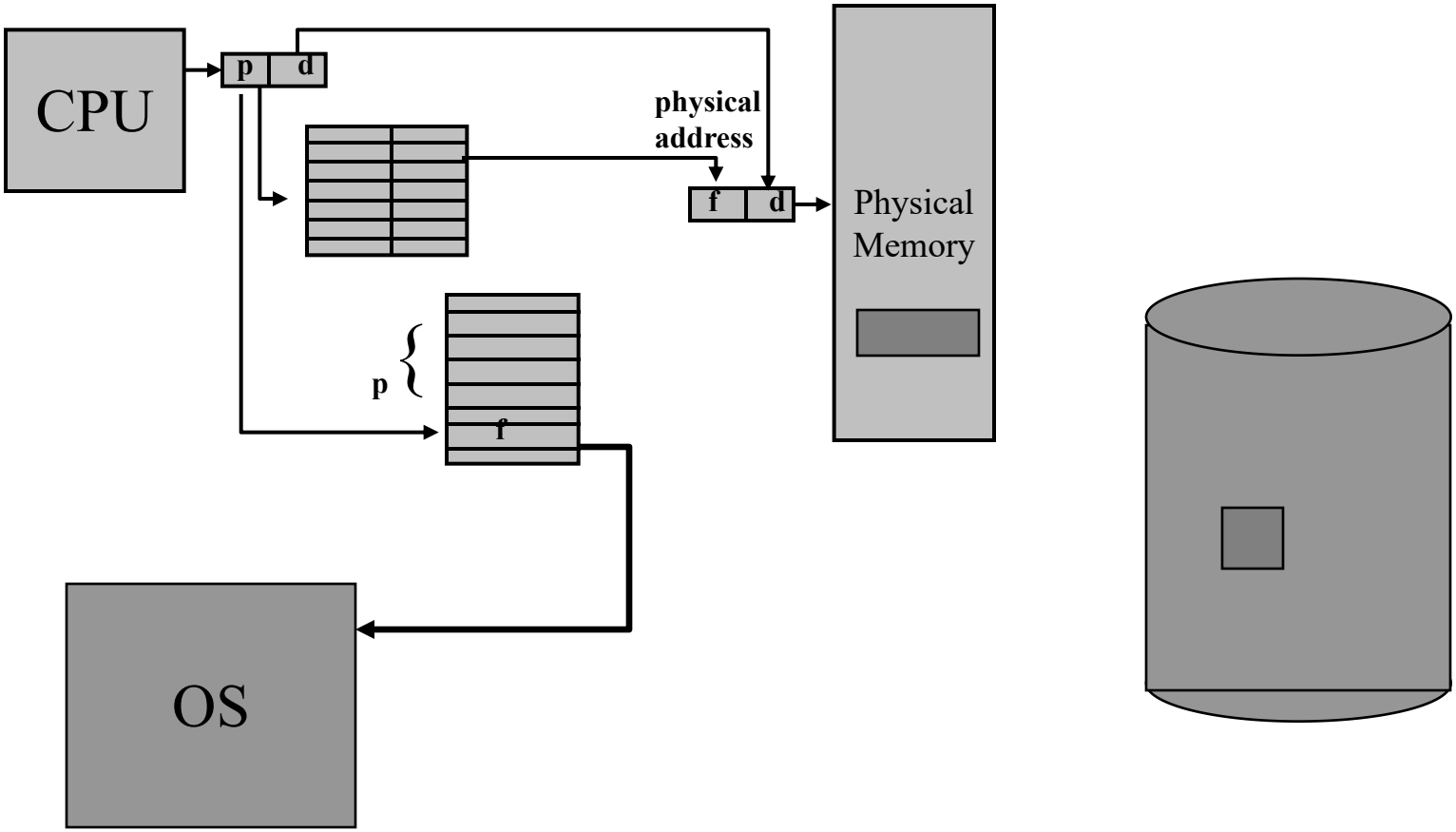
Page Fault Management



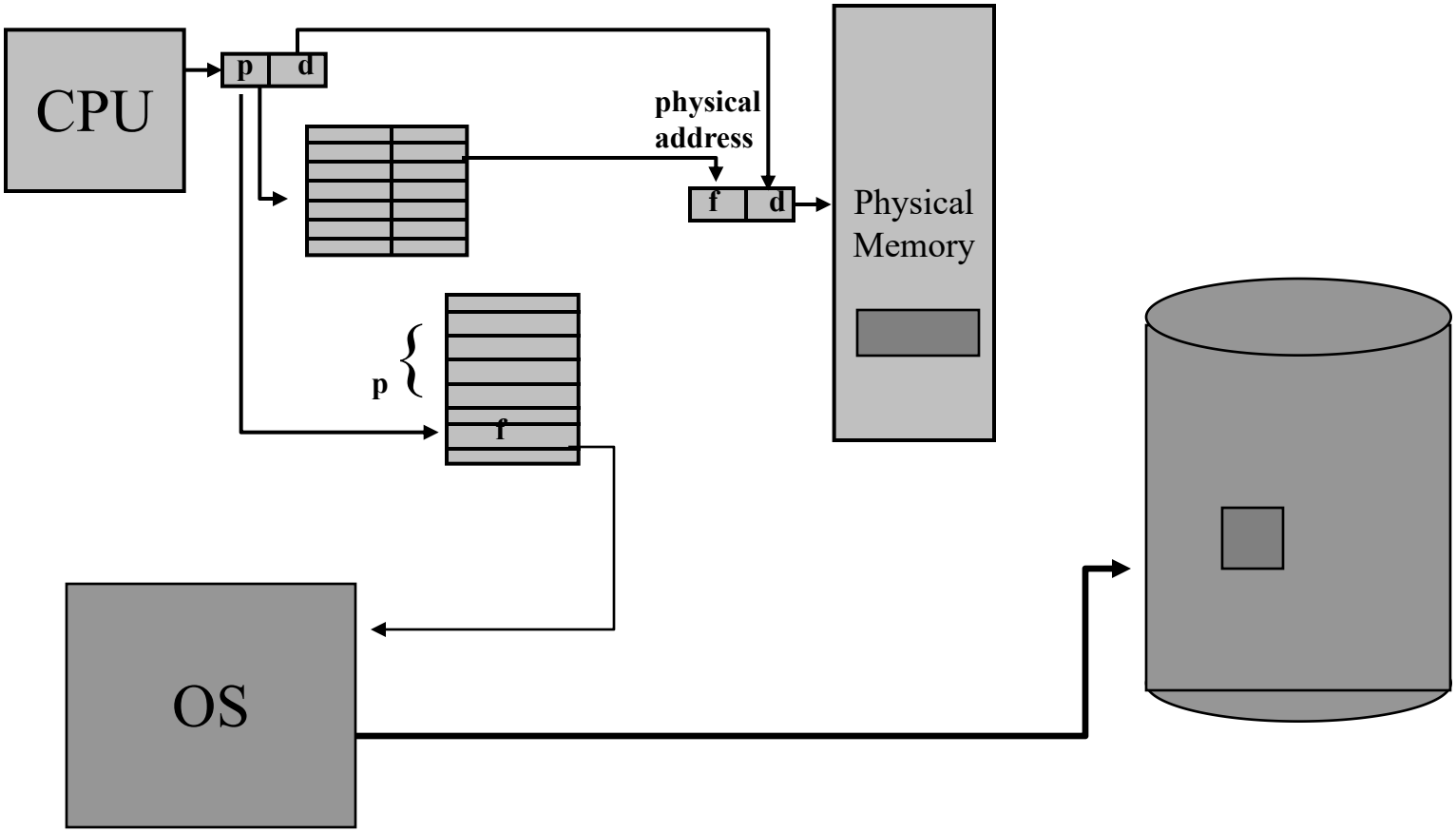
Page Fault Management



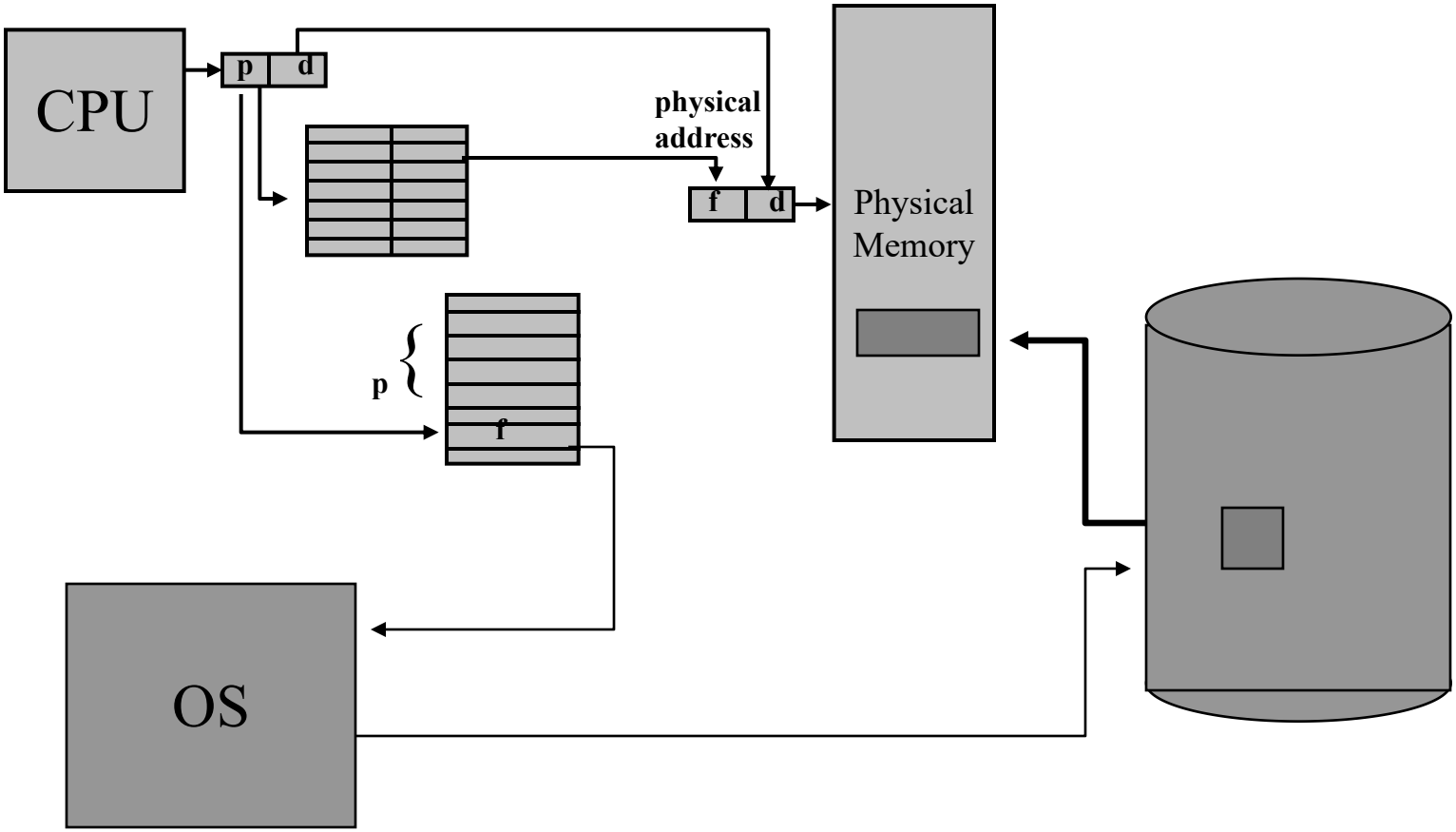
Page Fault Management



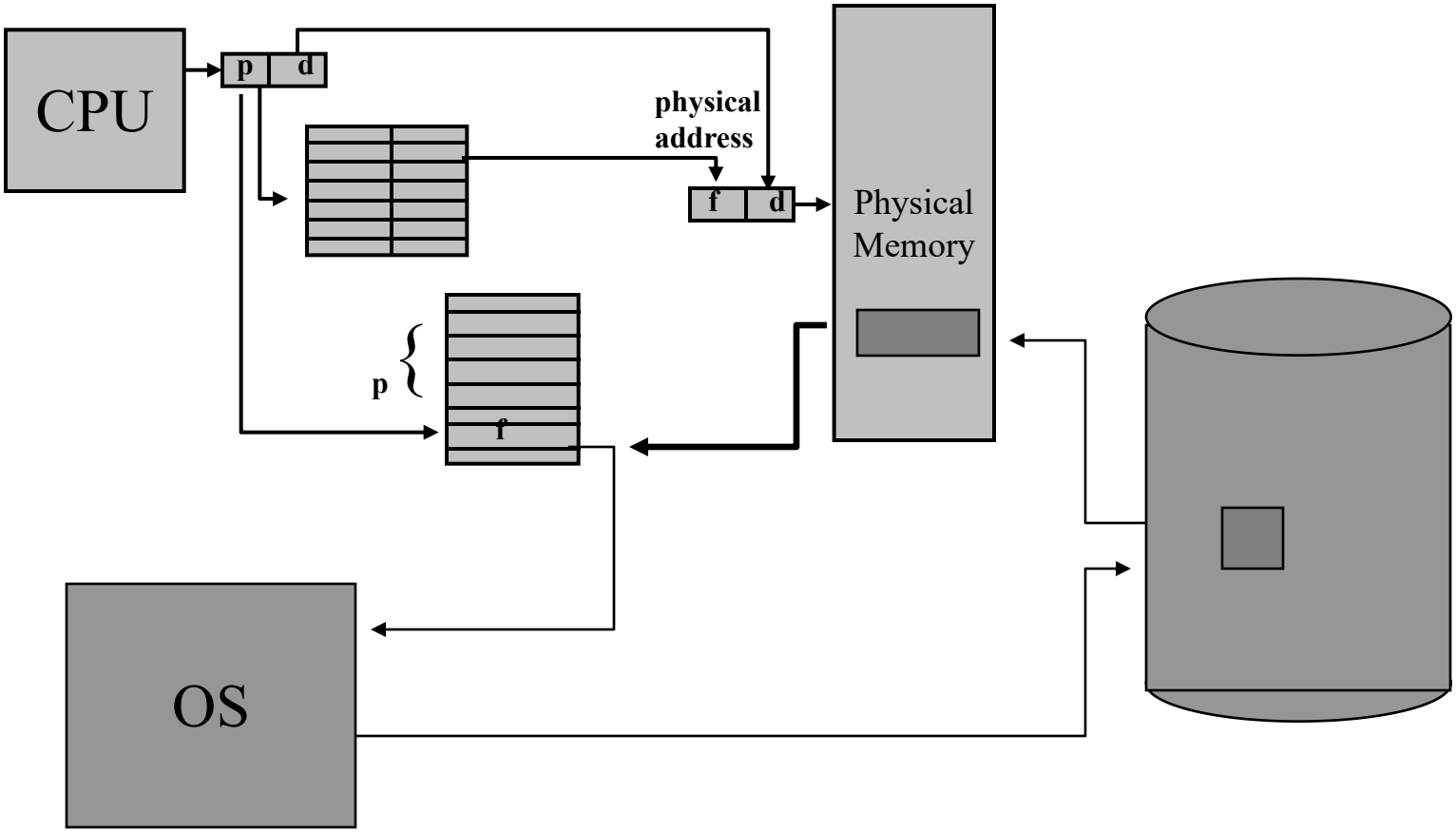
Page Fault Management



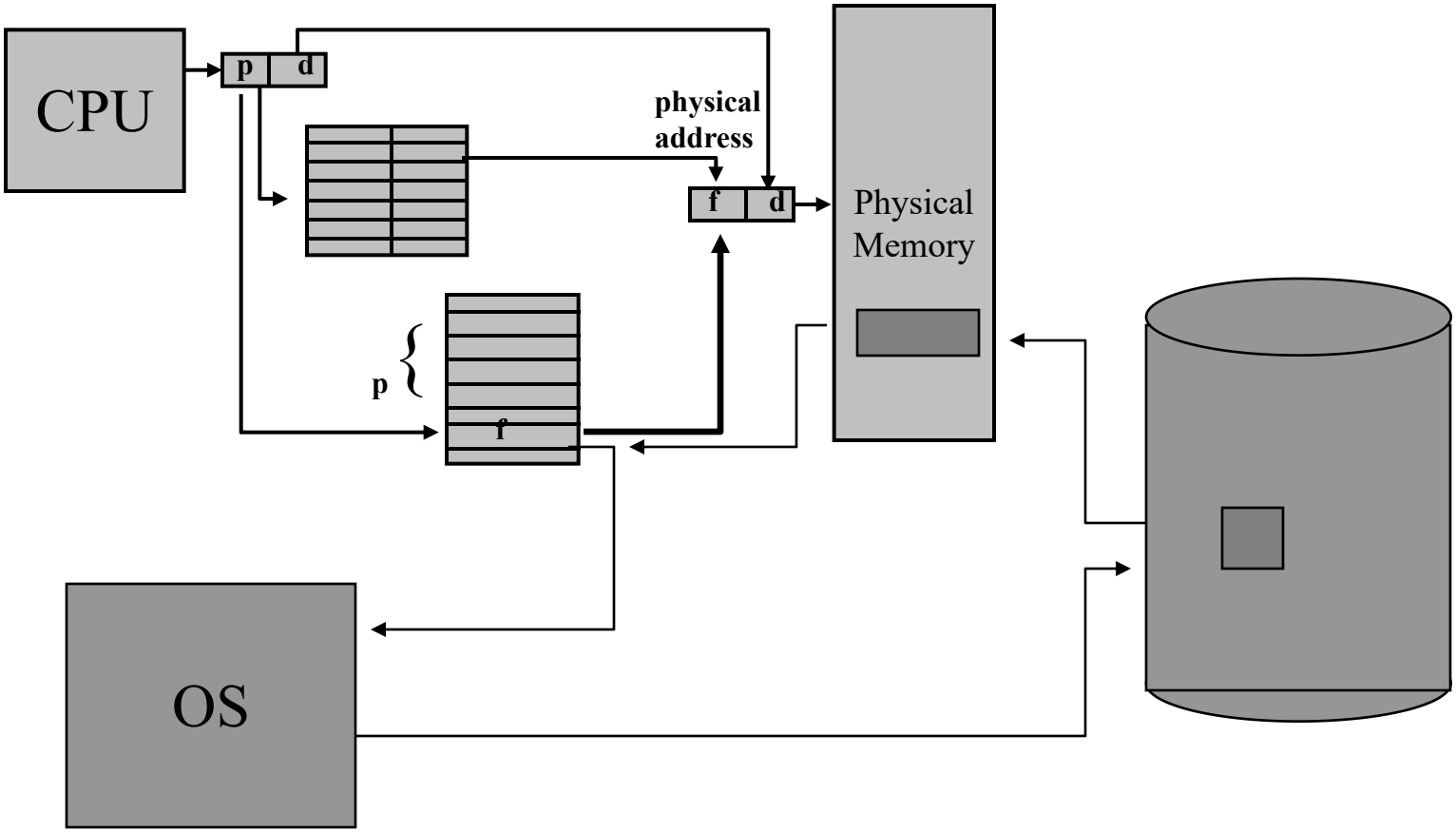
Page Fault Management



Page Fault Management



Page Fault Management



习题

3、请求分页存储管理系统，如果被访问页在内存，则满足一个内存请求需要200ns；如果被访问页不在内存，则分两种情况：若系统有空闲页框或被置换的页面未被修改，则置换所需要的时间为7ms；若系统被置换的页面已被修改，则置换所需要的时间为15ms；假设系统缺页率为5%，并且被换出的页面有60%是被修改，求有效访问时间（假设系统只运行一个进程且交换时CPU空闲）？

3.4 Page Replacement Algorithms

- **Algorithms**
  - Optimal Page Replacement algorithm ——OPT
  - Not recently used page replacement algorithm——NRU——use R bit and M bit
  - First-In, First-Out page replacement algorithm——FIFO
  - Second-Chance page replacement algorithm
  - Clock page replacement algorithm
  - Least Recently Used page replacement algorithm——LRU
  - Not Frequently Used page replacement algorithm——NFU
  - Working Set page replacement algorithm
  - WSClock page replacement algorithm

3.4 Page Replacement Algorithms

- **Summary of Algorithms**

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

3.4 Problem

- A computer has four page frames. The time of loading, time of last access, and the R and M bits for each page are as shown below(the times are in clock ticks):

Page	Loaded	Last ref.	R	M
0	126	279	0	0
1	230	260	1	0
2	120	272	1	1
3	160	280	1	1

(a) Which page will NRU replace?

(b) Which page will FIFO replace?

(c) Which page will LRU repalce?

(d) Which page will second chance replace?

3.5 Design issues

- **Local versus Global Allocation Policies**

	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.

3.5 Design issues

- **Load control**
  - Thrashing
  - The degree of multiprogramming
- **Page size**
  - The best page size

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

  - P: the page size (bytes)
  - s: the average process size (bytes)
  - e: the page entry size (bytes)

Setting Page Fault Rates

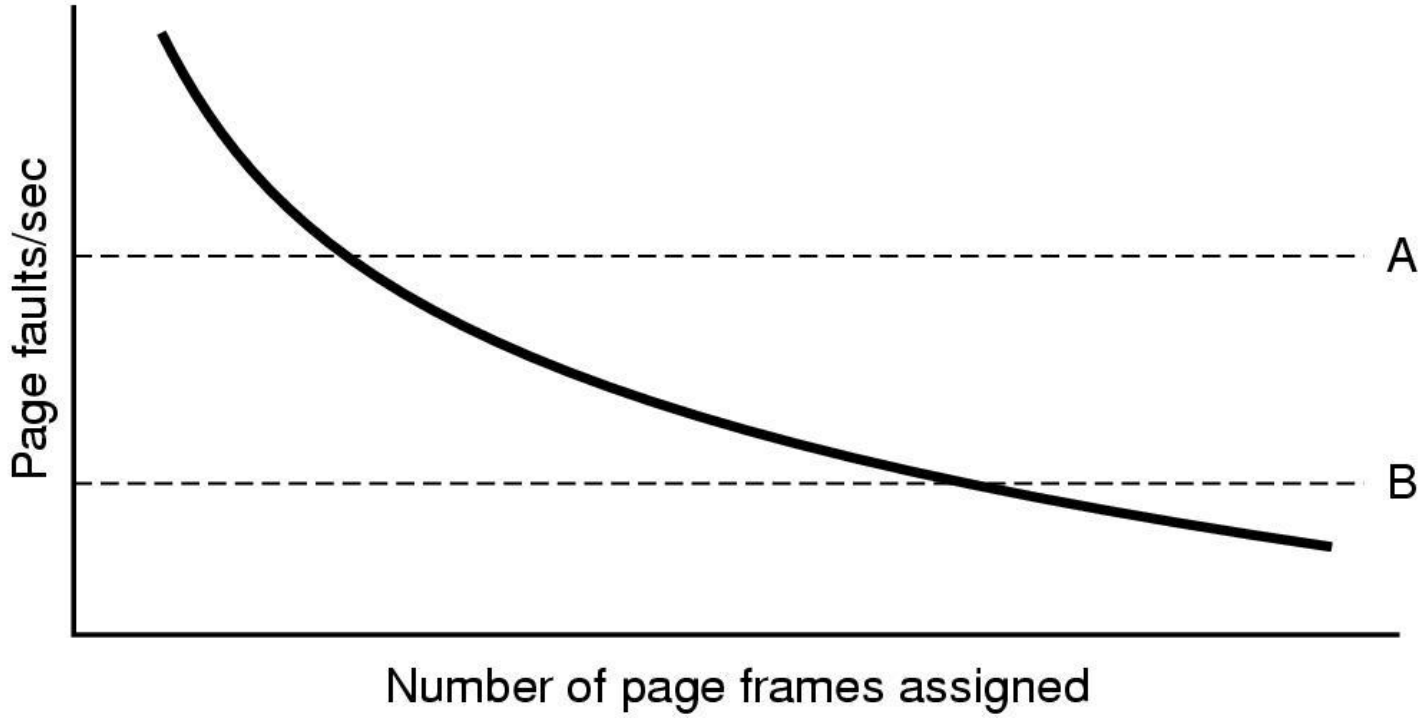


Figure 3-24. Page fault rate as a function of the number of page frames assigned.

3.5 Design issues

- **Separate Instruction and Data Spaces**

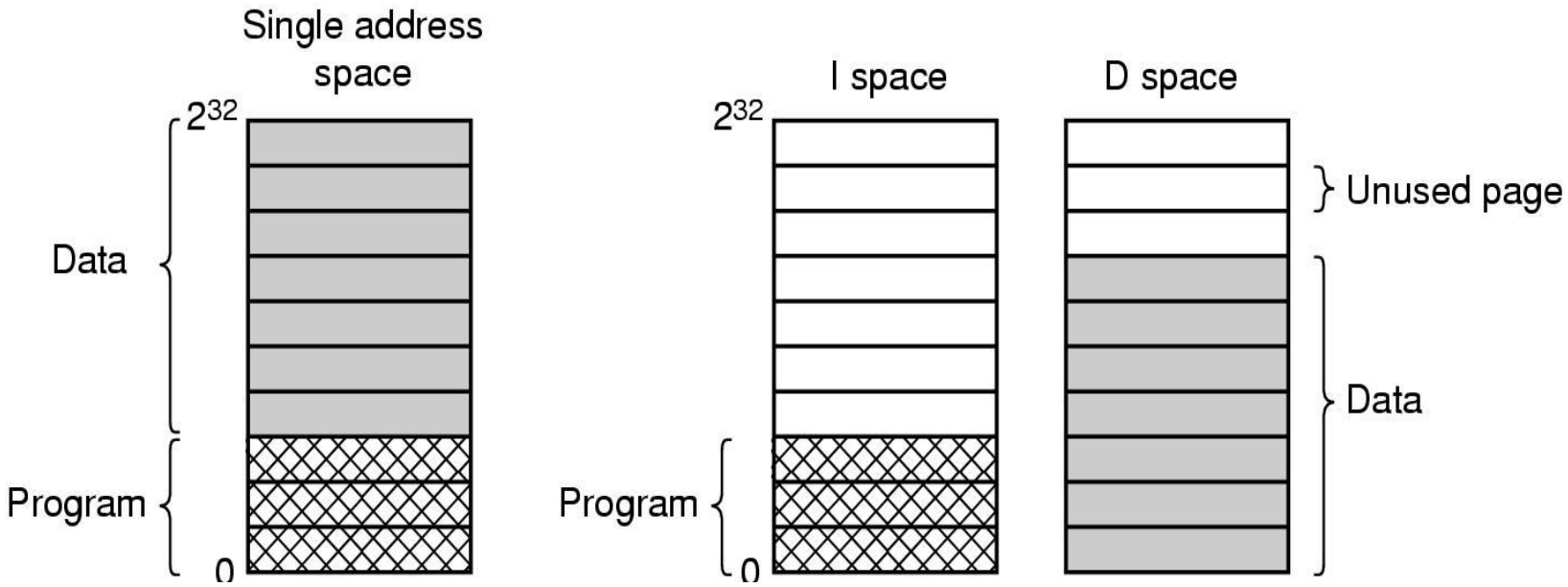


Figure 3-25. (a) One address space.
(b) Separate I and D spaces.

3.5 Design issues

- **Shared Pages**

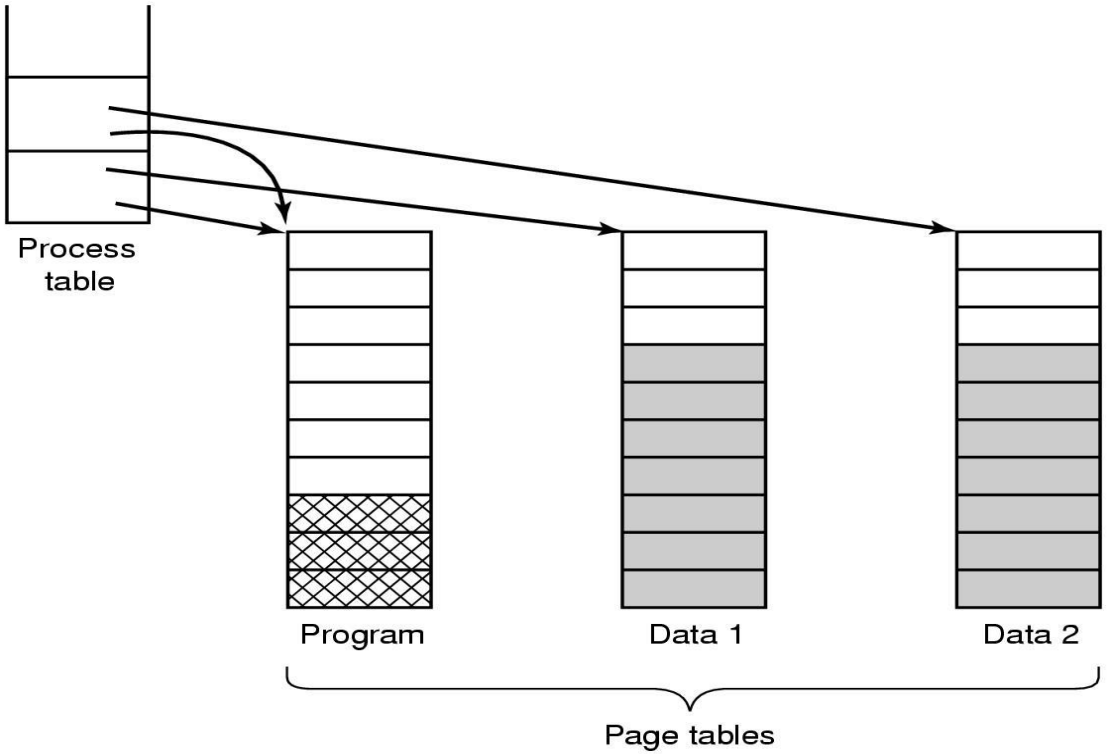


Figure 3-26. Two processes sharing the same program sharing its page table.

3.5 Design issues

- **Shared Libraries**

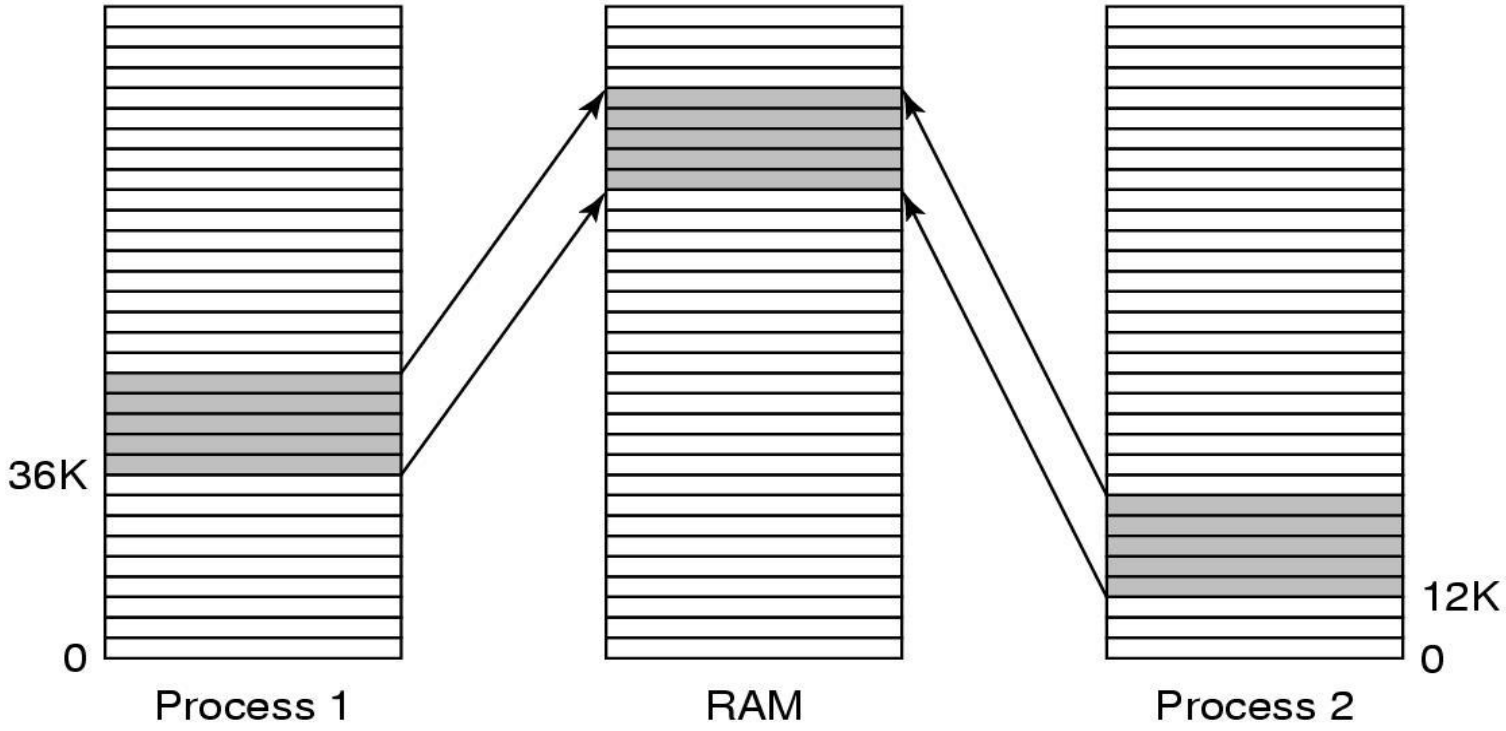


Figure 3-27. A shared library being used by two processes.

3.5 Design issues

- **Mapped Files**
  - An alternative model for I/O
  - For shared libraries
- **Cleaning Policy**
  - Page daemon
  - Method: two handed clock
- **Virtual Memory Interface**
  - Virtual memory is transparent to processes and programmers
  - For distributed shared memory

3.6 Implementation issues

- **OS involvement with Paging**
- **Page Fault Handling**
- **Instruction backup**
- **Locking pages in memory**
- **Backing store**
- **Separation of policy and mechanism**

Page Fault Handling (1)

- The hardware traps to the kernel, saving the program counter on the stack.
- An assembly code routine is started to save the general registers and other volatile information.
- The operating system discovers that a page fault has occurred and tries to discover which virtual page is needed.
- 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)

- If the page frame selected is dirty, the page is scheduled for transfer to the disk, and a context switch takes place.
- 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.
- When disk interrupt indicates page has arrived, page tables updated to reflect position, frame marked as being in normal state.

Page Fault Handling (3)

- Faulting instruction backed up to state it had when it began and program counter reset to point to that instruction.
- Faulting process scheduled, operating system returns to the (assembly language) routine that called it.
- This routine reloads registers and other state information and returns to user space to continue execution, as if no fault had occurred.

Instruction Backup

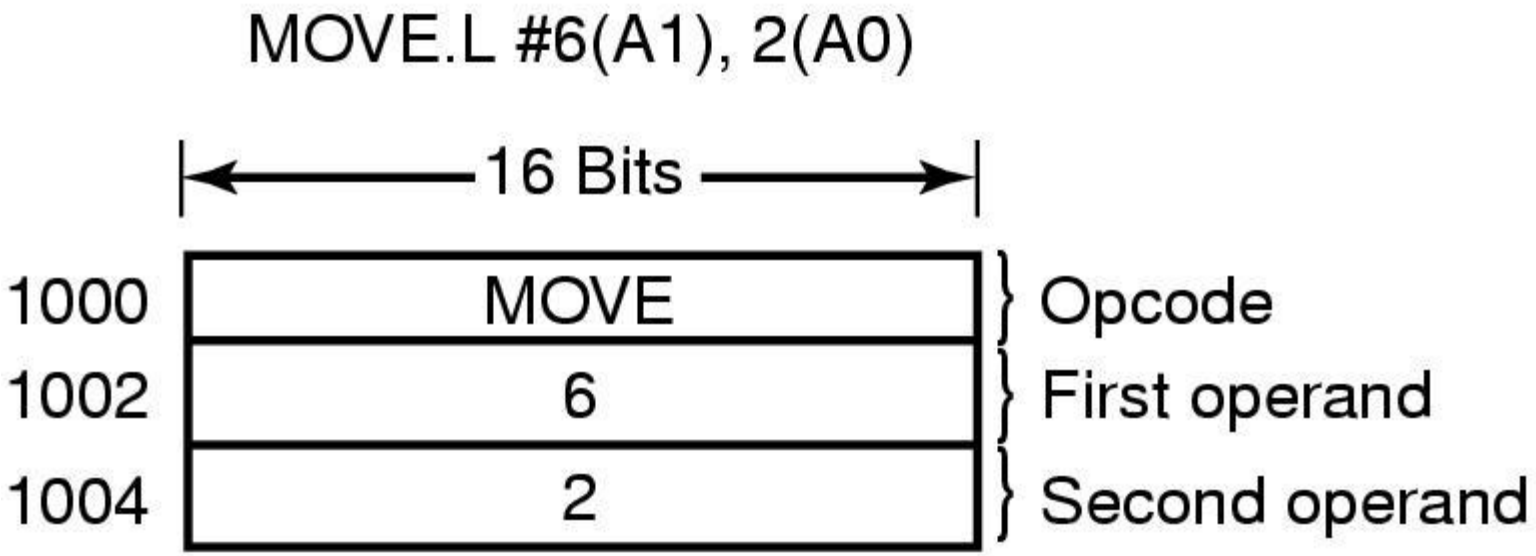


Figure 3-28. An instruction causing a page fault.

Backing Store (1)

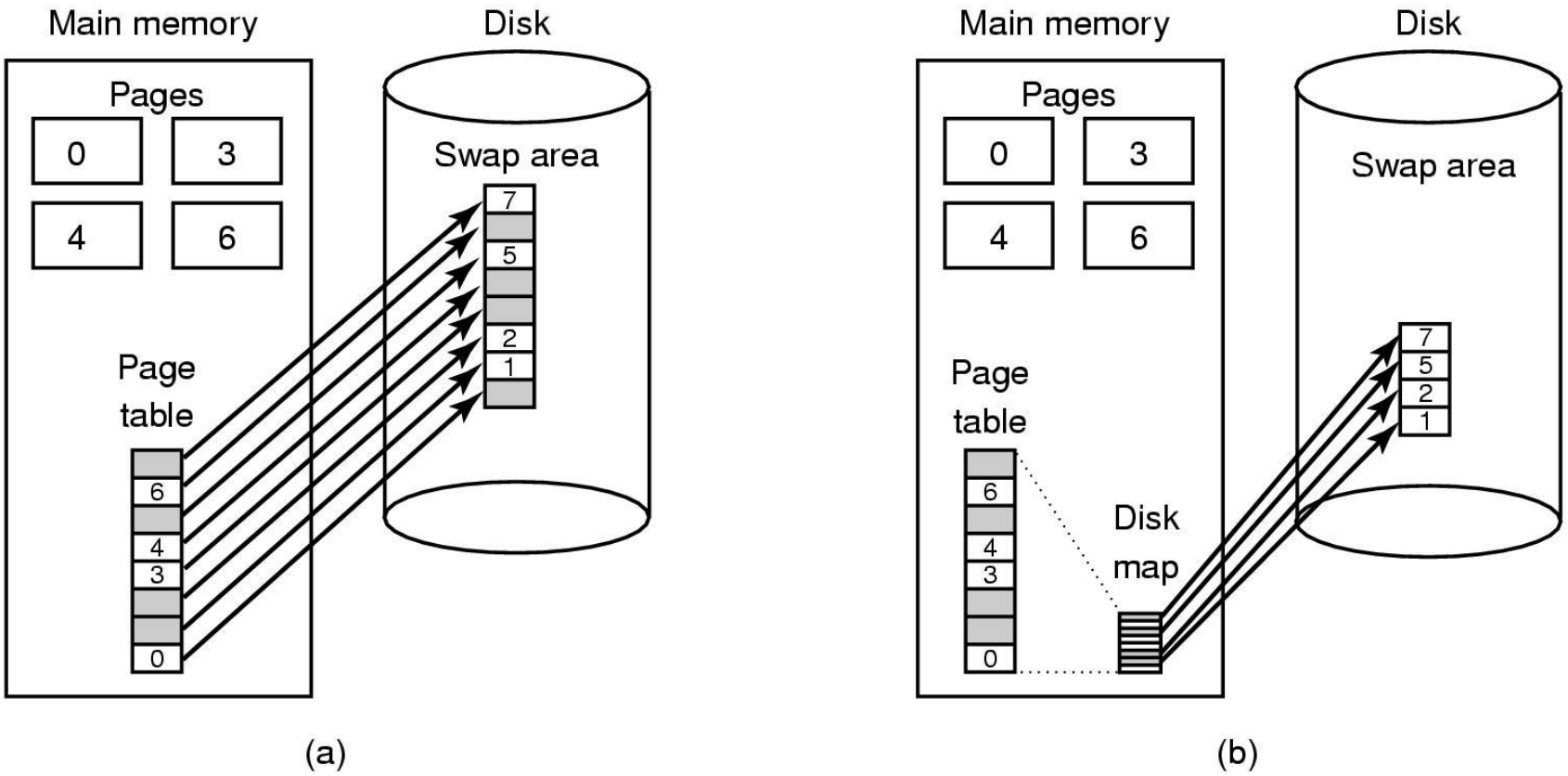


Figure 3-29. (a) Paging to a static swap area.

Backing Store (2)

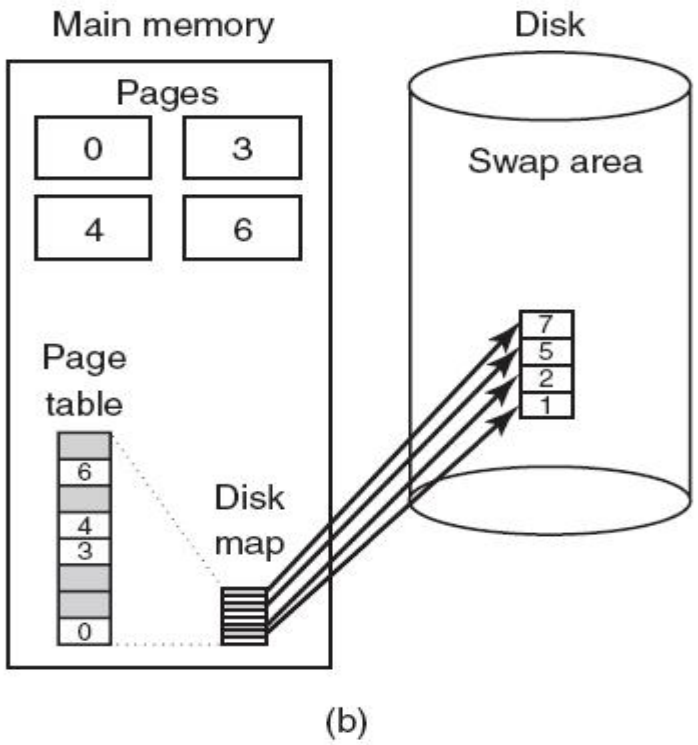
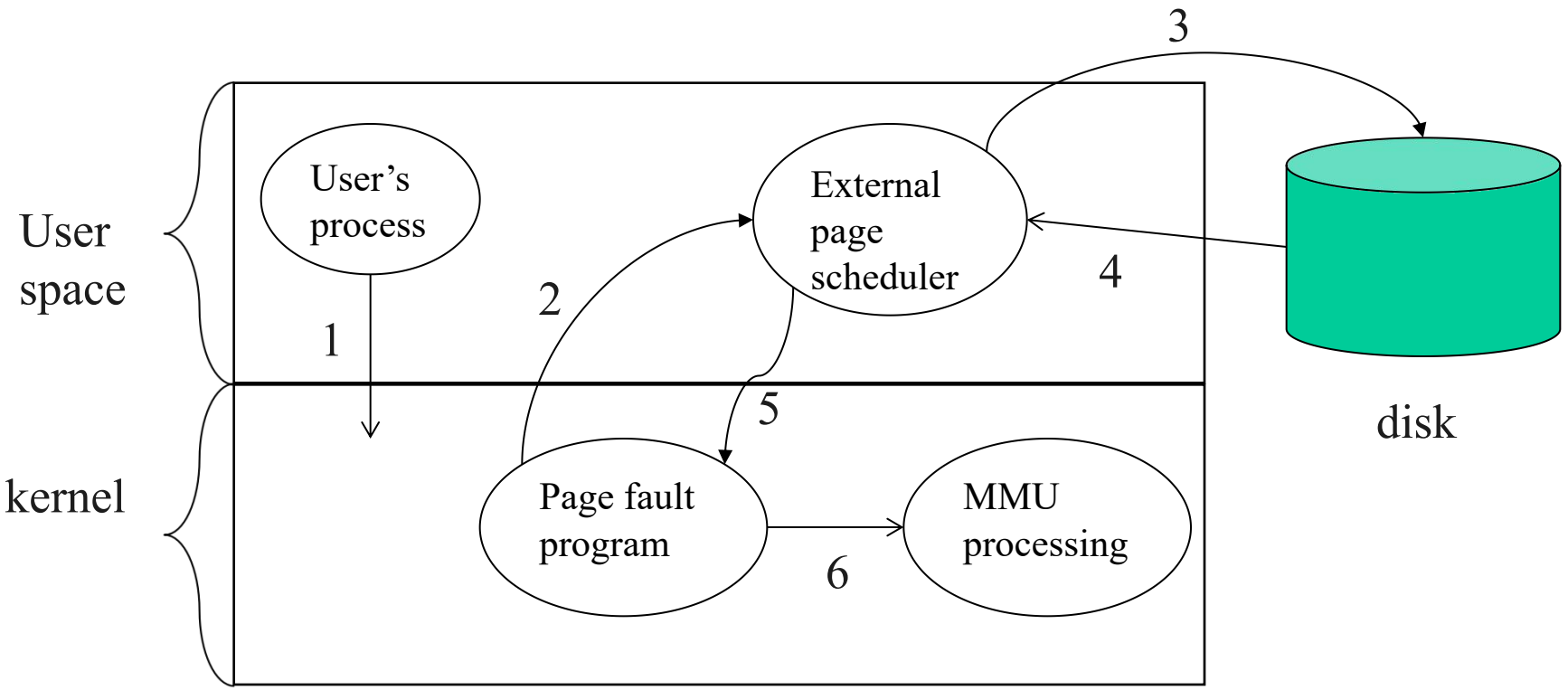


Figure 3-29. (b) Backing up pages dynamically.

Separation of policy and mechanism



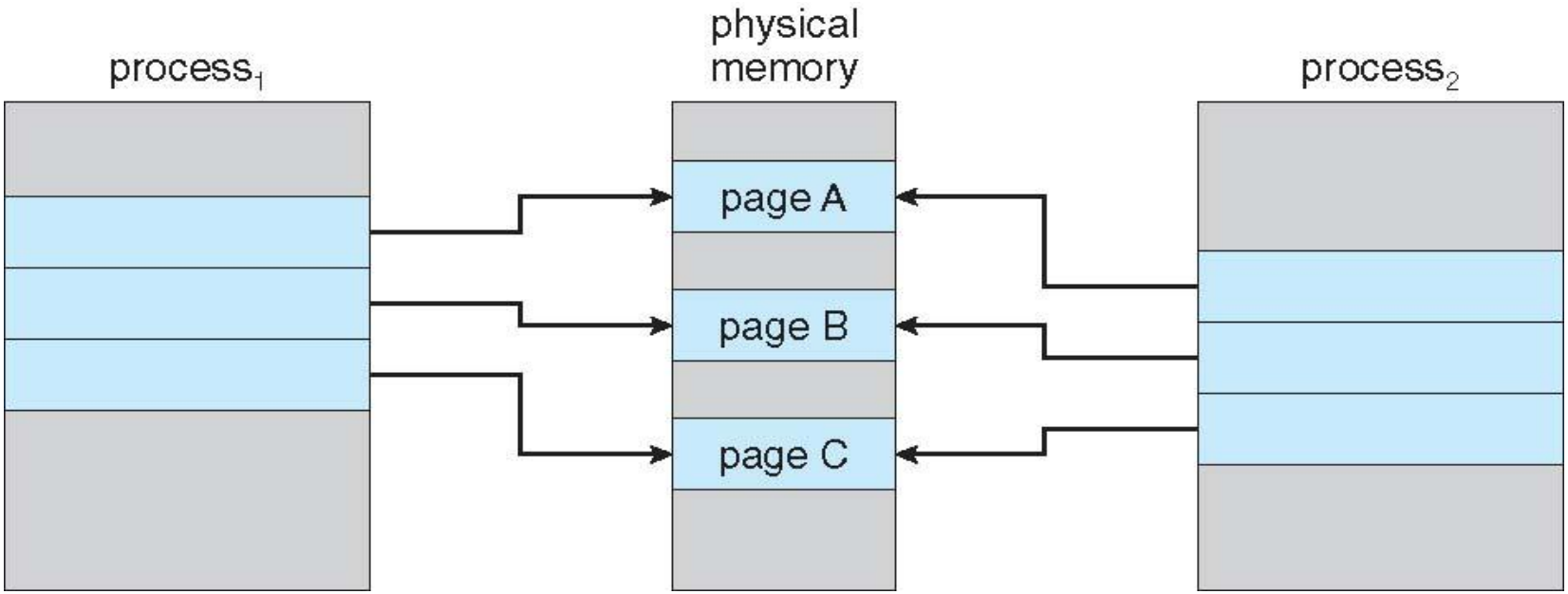
Demand Paging Optimizations

- Swap space I/O faster than file system I/O even if on the same device
  - Swap allocated in larger chunks, less management needed than file system
- Copy entire process image to swap space at process load time
  - Then page in and out of swap space
  - Used in older BSD Unix
- Demand page in from program binary on disk, but discard rather than paging out when freeing frame
  - Used in Solaris and current BSD
  - Still need to write to swap space
    - Pages not associated with a file (like stack and heap) – **anonymous memory**
    - Pages modified in memory but not yet written back to the file system
- Mobile systems
  - Typically don't support swapping
  - Instead, demand page from file system and reclaim read-only pages (such as code)

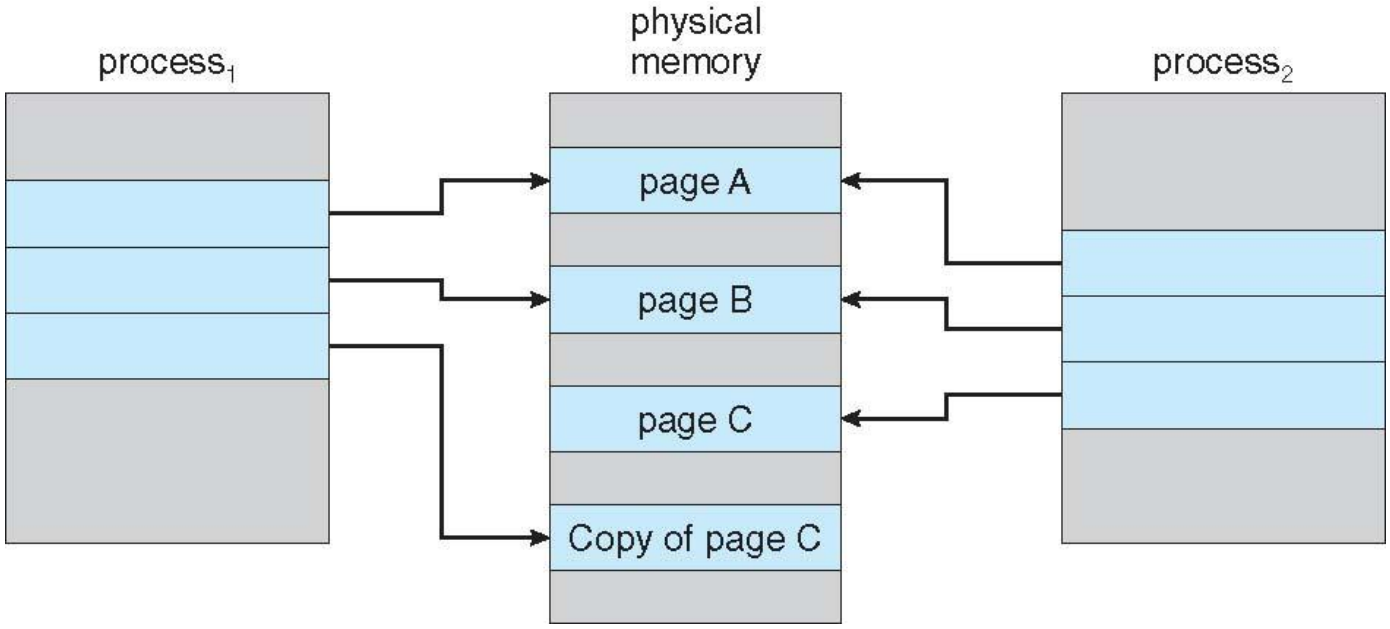
Copy-on-Write

- **Copy-on-Write** (COW) allows both parent and child processes to initially ***share*** the same pages in memory
  - If either process modifies a shared page, only then is the page copied
- COW allows more efficient process creation as only modified pages are copied
- In general, free pages are allocated from a **pool** of **zero-fill-on-demand** pages
  - Pool should always have free frames for fast demand page execution
    - Don't want to have to free a frame as well as other processing on page fault
  - Why zero-out a page before allocating it?
- `vfork()` variation on `fork()` system call has parent suspend and child using copy-on-write address space of parent
  - Designed to have child call `exec()`
  - Very efficient

Before Process 1 Modifies Page C



After Process 1 Modifies Page C



Buddy System

Allocates memory from fixed-size segment consisting of physically-contiguous pages

Memory allocated using **power-of-2 allocator**

- Satisfies requests in units sized as power of 2
- Request rounded up to next highest power of 2
- When smaller allocation needed than is available, current chunk split into two buddies of next-lower power of 2
  - Continue until appropriate sized chunk available

For example, assume 256KB chunk available, kernel requests 21KB

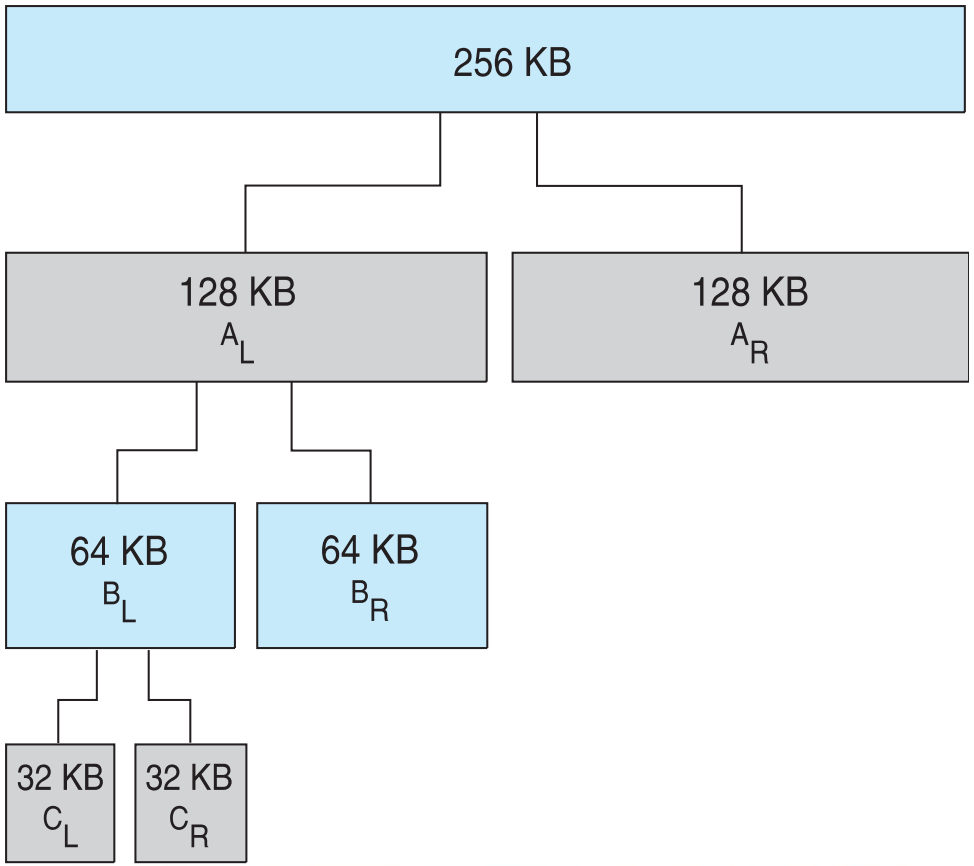
- Split into A_L and A_R of 128KB each
  - One further divided into B_L and B_R of 64KB
    - One further into C_L and C_R of 32KB each – one used to satisfy request

Advantage – quickly **coalesce** unused chunks into larger chunk

Disadvantage - fragmentation

Buddy System Allocator

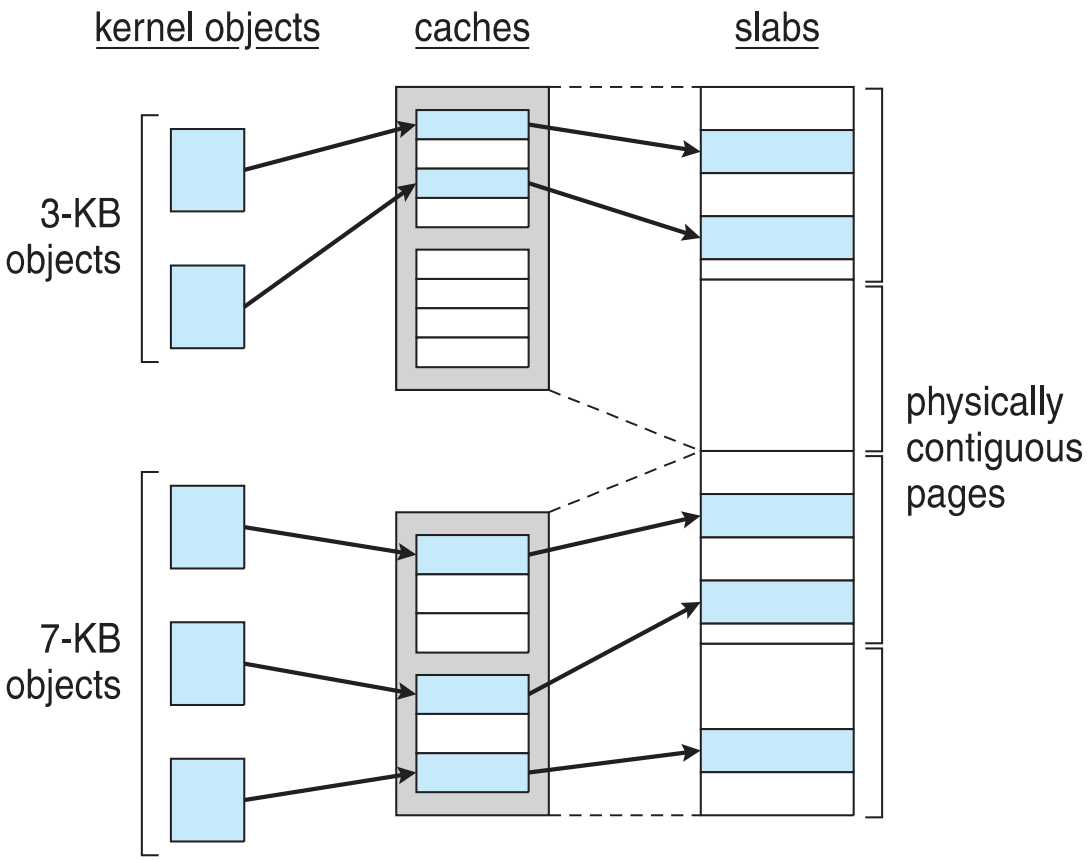
physically contiguous pages



Slab Allocator

- Alternate strategy
- **Slab** is one or more physically contiguous pages
- **Cache** consists of one or more slabs
- Single cache for each unique kernel data structure
  - Each cache filled with **objects** – instantiations of the data structure
- When cache created, filled with objects marked as **`free`**
- When structures stored, objects marked as **`used`**
- If slab is full of used objects, next object allocated from empty slab
  - If no empty slabs, new slab allocated
- Benefits include no fragmentation, fast memory request satisfaction

Slab Allocation



Slab Allocator in Linux

- For example process descriptor is of type `struct task_struct`
- Approx 1.7KB of memory
- New task -> allocate new struct from cache
  - Will use existing free `struct task_struct`
- Slab can be in three possible states
  - Full – all used
  - Empty – all free
  - Partial – mix of free and used
- Upon request, slab allocator
  - Uses free struct in partial slab
  - If none, takes one from empty slab
  - If no empty slab, create new empty

Slab Allocator in Linux (Cont.)

Slab started in Solaris, now wide-spread for both kernel mode and user memory in various OSes

Linux 2.2 had SLAB, now has both SLOB and SLUB allocators

- SLOB for systems with limited memory
  - Simple List of Blocks – maintains 3 list objects for small, medium, large objects
- SLUB is performance-optimized SLAB removes per-CPU queues, metadata stored in page structure

3.7 Segmentation

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

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

3.7 Segmentation

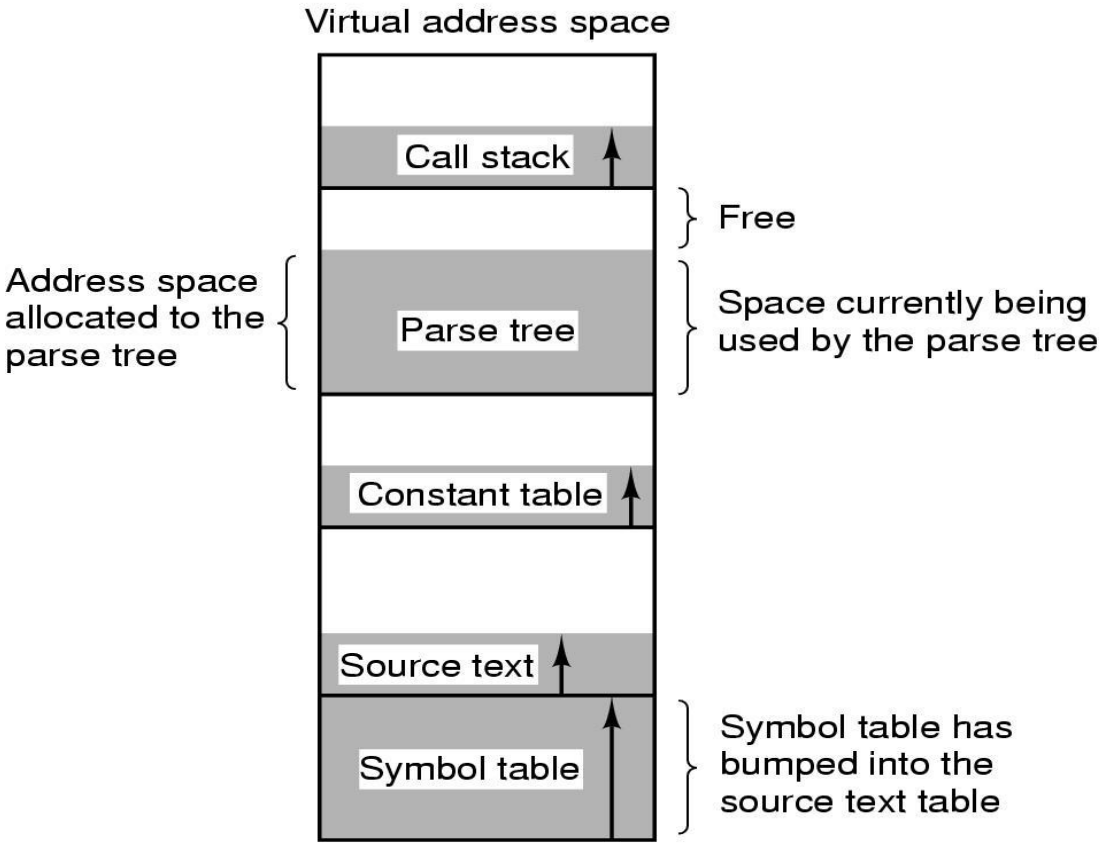


Figure 3-31. In a one-dimensional address space with growing tables, one table may bump into another.

3.7 Segmentation

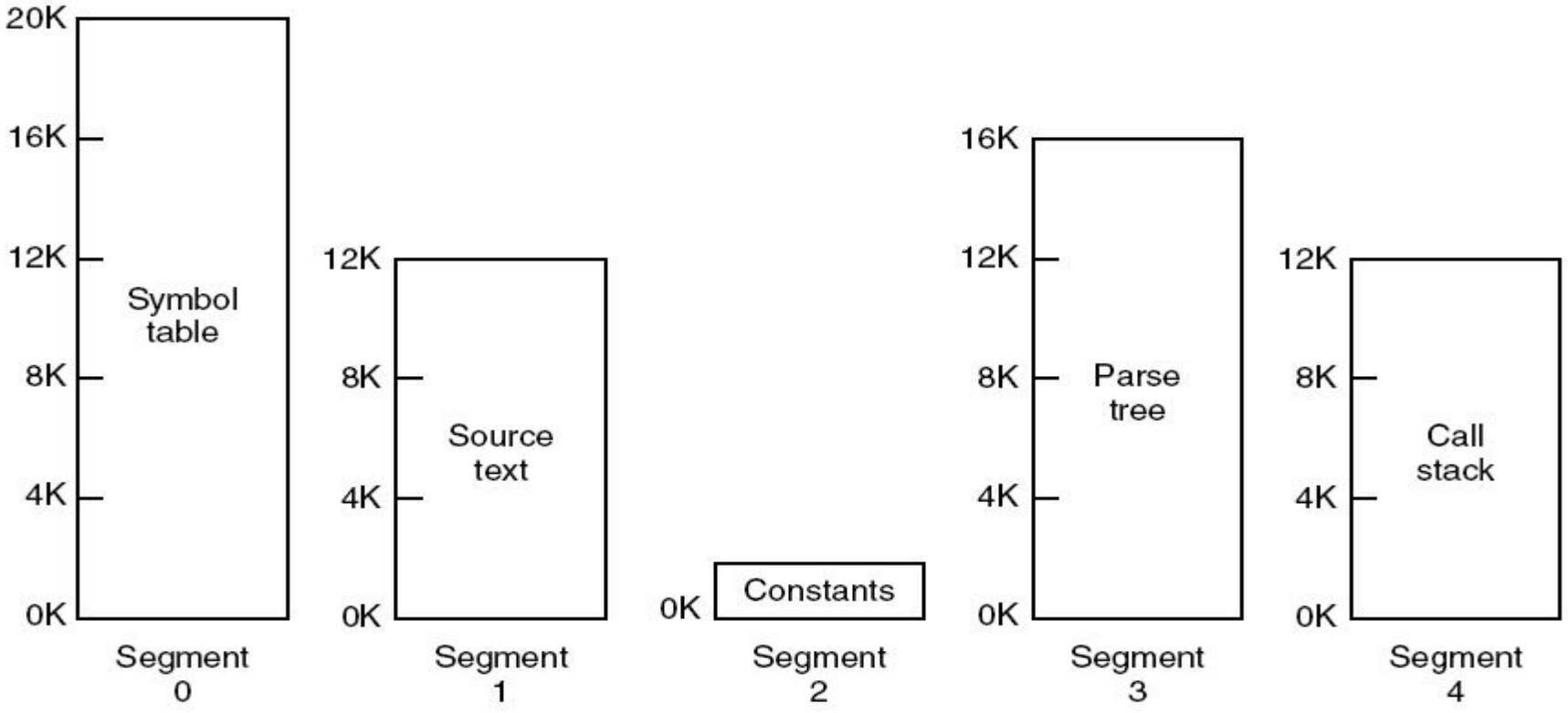


Figure 3-32. A segmented memory allows each table to grow or shrink independently of the other tables.

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)

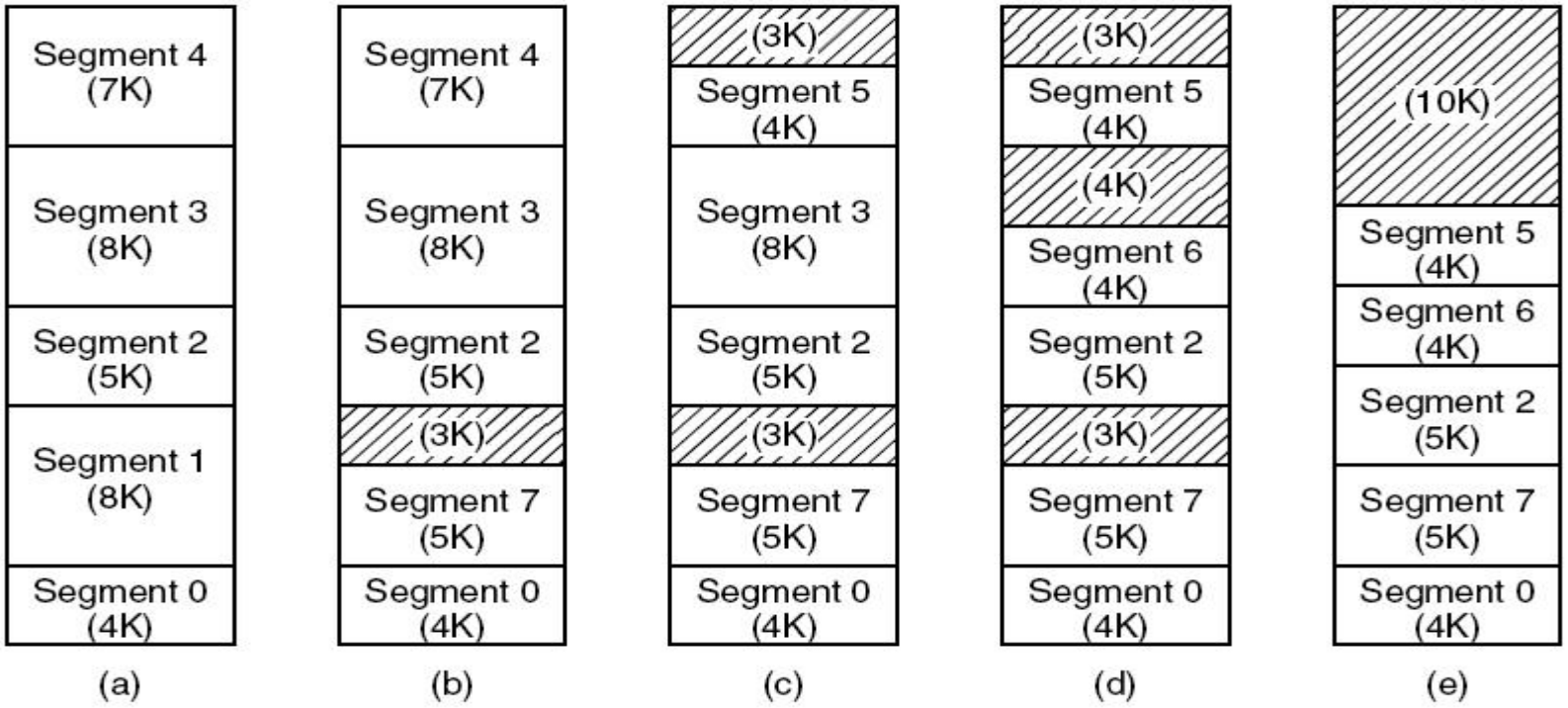


Figure 3-34. (a)-(d) Development of checkerboarding. (e) Removal of the checkerboarding by compaction.

Segmentation with Paging: MULTICS (2)

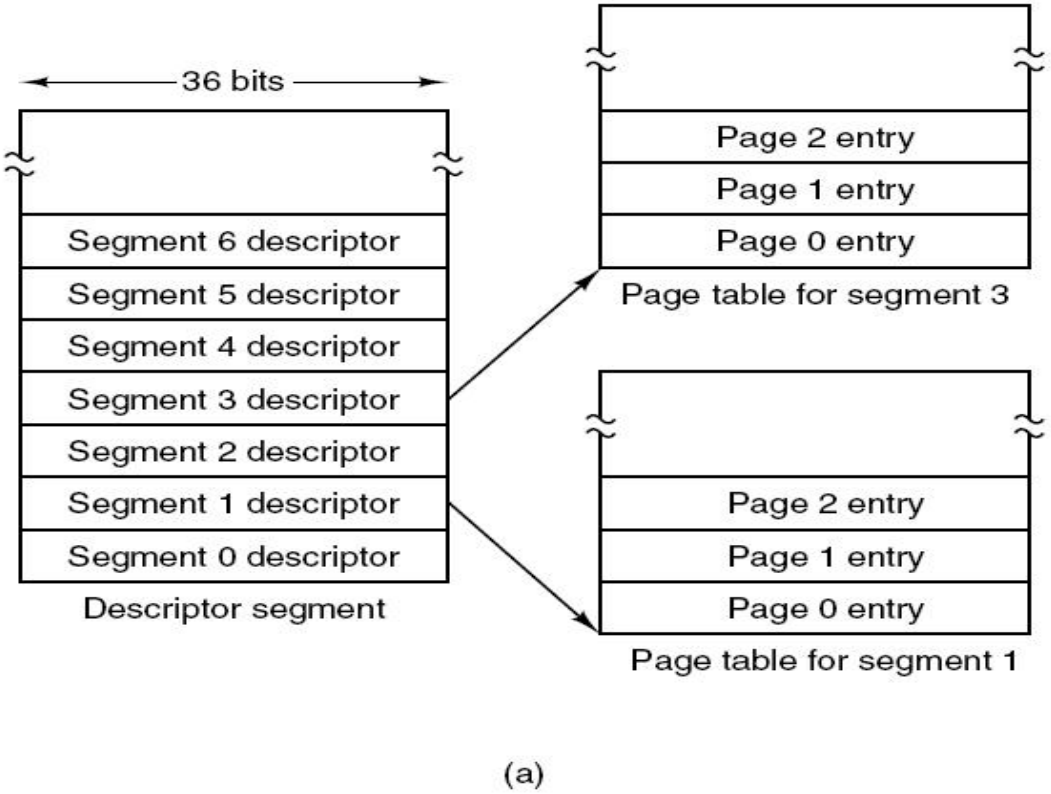


Figure 3-35. The MULTICS virtual memory. (a) The descriptor segment points to the page tables.

Segmentation with Paging: MULTICS (5)

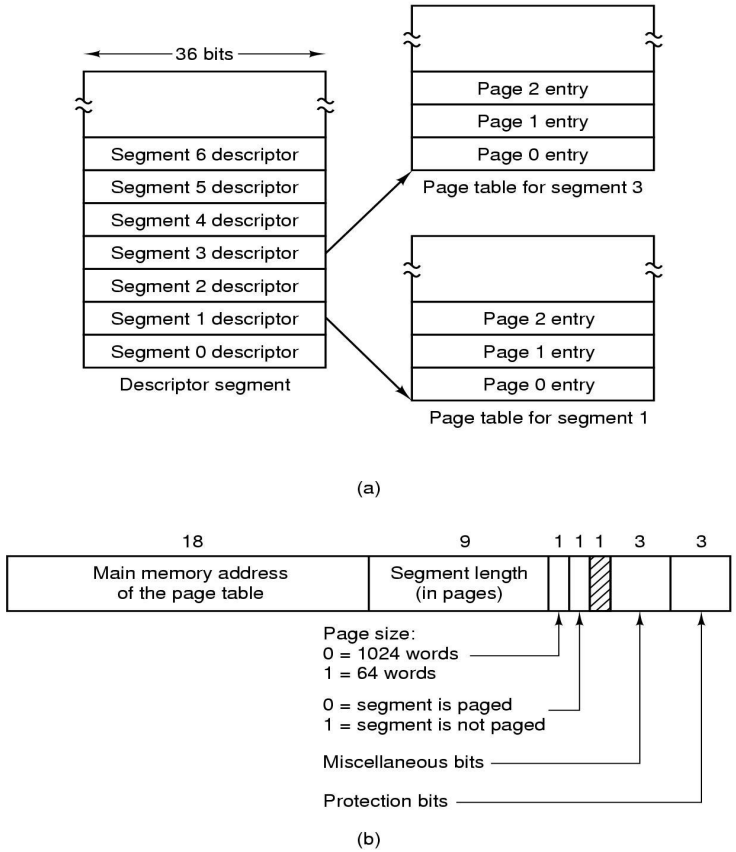


Figure 3-35. The MULTICS virtual memory. (b) A segment descriptor. The numbers are the field lengths.

Segmentation with Paging: MULTICS (6)

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

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

Segmentation with Paging: MULTICS (7)

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

Segmentation with Paging: MULTICS (8)

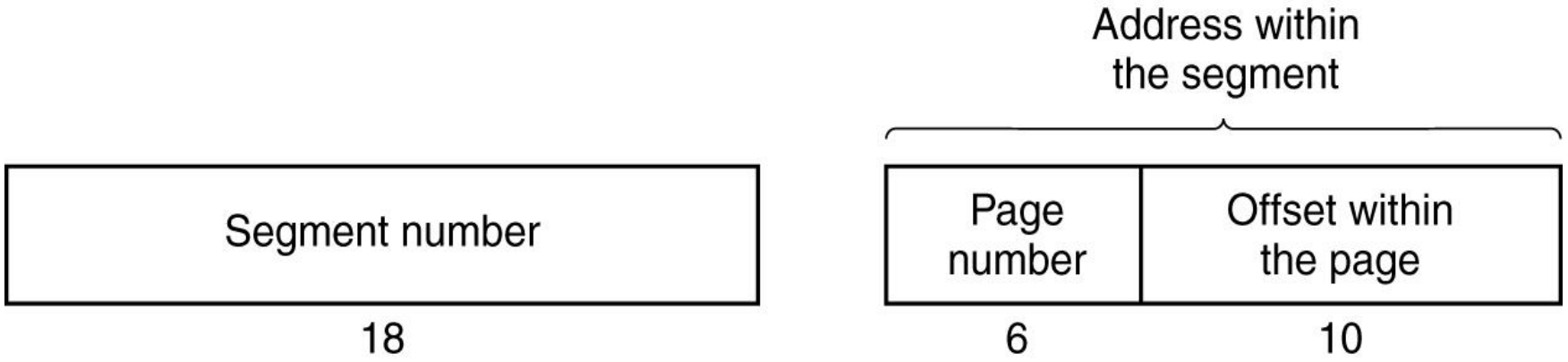


Figure 3-36. A 34-bit MULTICS virtual address.

Segmentation with Paging: MULTICS (9)

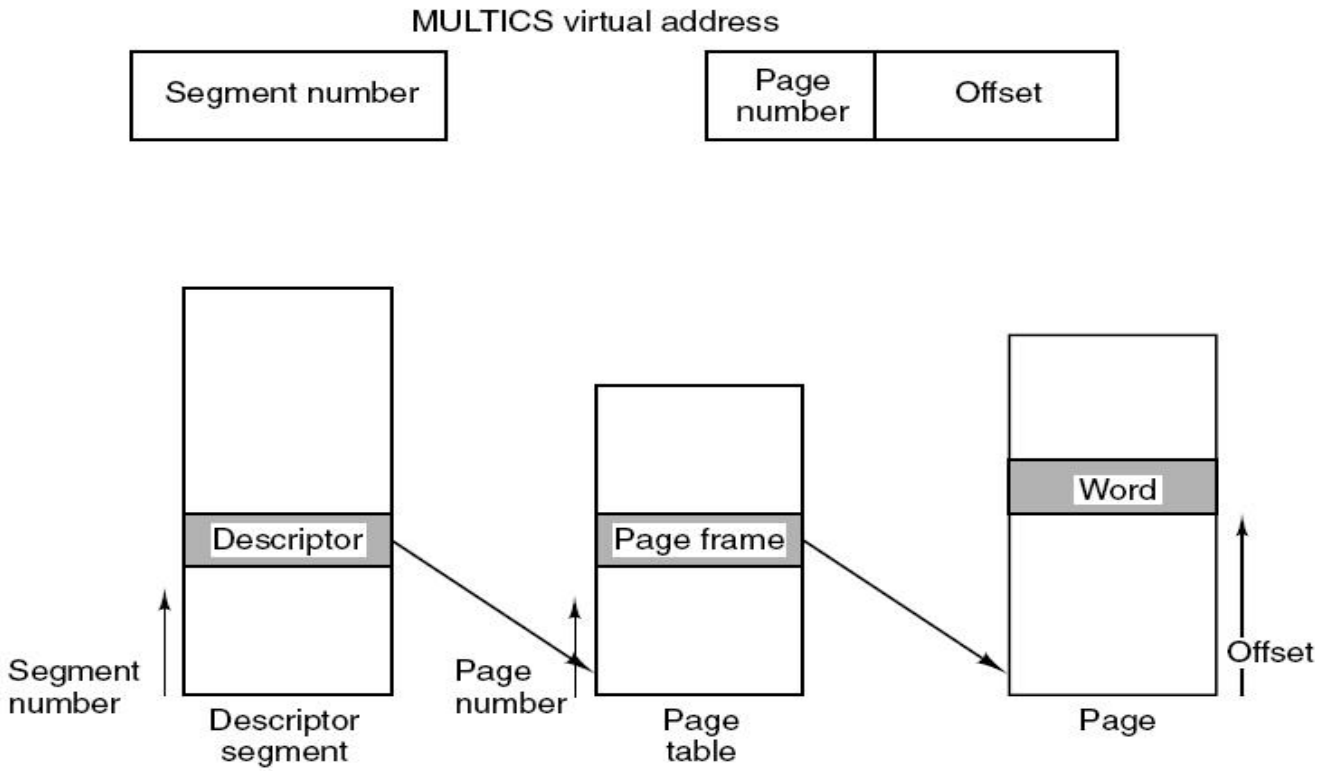


Figure 3-37. Conversion of a two-part MULTICS address into a main memory address.

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.

Summary

- Frame Allocation Policies
  - Local vs. Global
  - Instruction vs. data
- Page Fault Management
- Segmentation

Questions

- What is the effective memory access time in a system with a memory access time of 20 ns, a disk access time of 8 ms, and a page fault rate of $1*10^{-6}$?
- Programs today are so large, and memory is small (relative to the size of all of the programs needed at one time). A paging system needs to ensure that pages needed by the CPU are usually in main memory. Why? (Include in your explanation the penalty that a program suffers if the page is not in main memory). What mechanism in normal programs makes paging an acceptable solution? (Why don't we have many page faults?)
- What is the difference between paging and segmentation? What are the advantages and disadvantages of each?
- How can OSs use a pure paging scheme in CPUs that implement segmentation?

谢 谢！