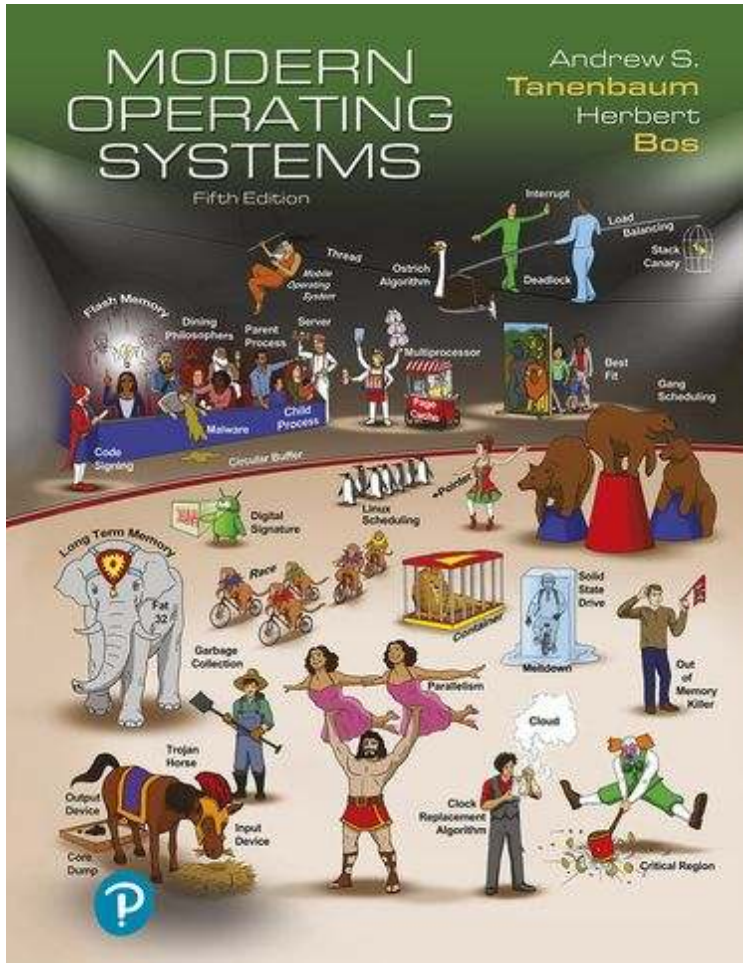
Modern Operating Systems

Fifth Edition



Chapter 3

Memory Management

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Memory

Paraphrase of Parkinson's Law, **"Programs expand to fill the memory available to hold them."**

My laptop with 32GB of RAM has 20,000 times more memory than the IBM 7094 (32768 words of 36 bits)—the largest computer in the world in the early 1960s

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Memory

- Memory Abstractions
- Virtual Memory
- Page Replacement Algorithms
- Design Issues for Paging Systems

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

No Memory Abstraction: Monoprogramming

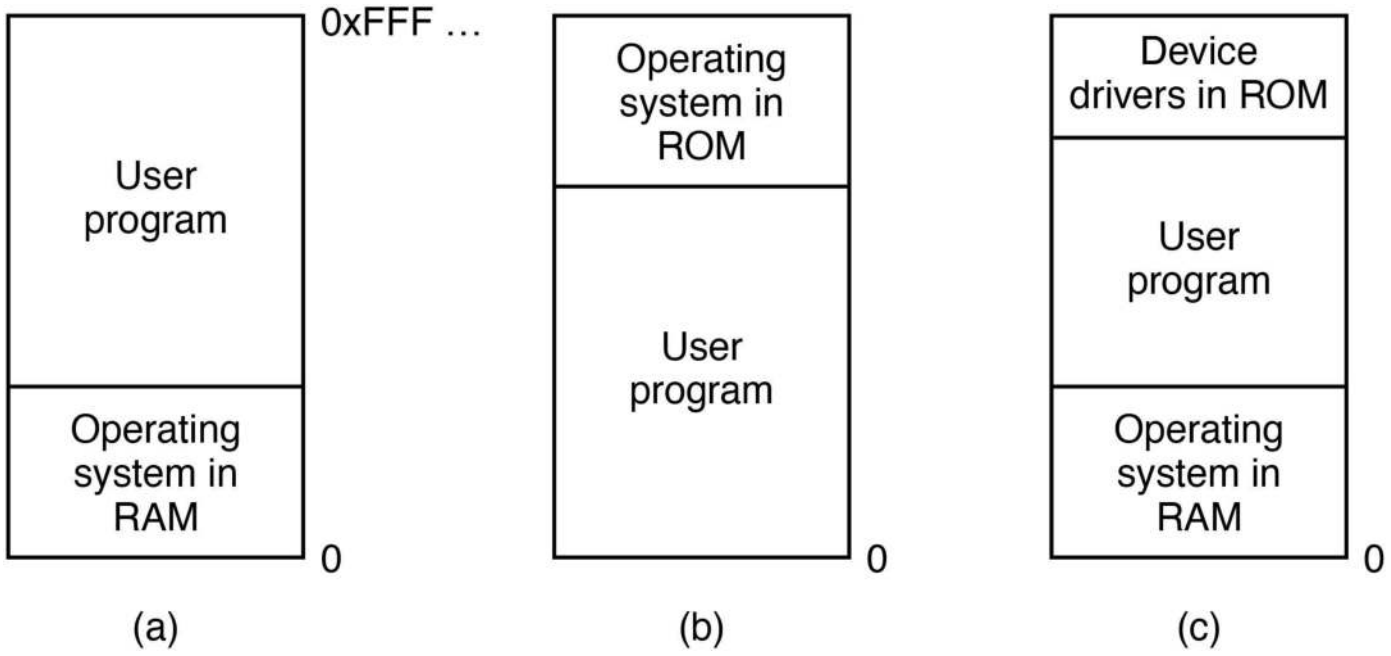
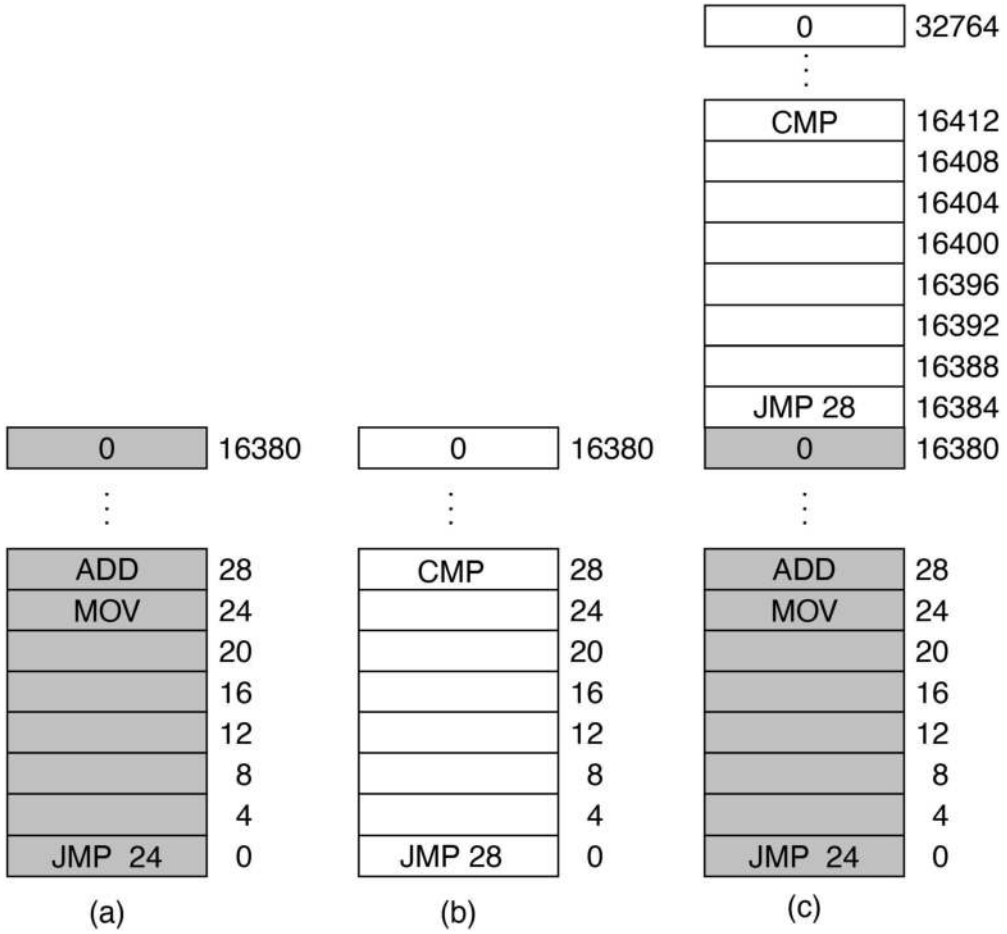


Figure 3-1. Three simple ways of organizing memory with an operating system and one user process.

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

No Memory Abstraction: Multiprogramming



Naive approach:

Move each program to a dedicated region

Figure 3-2. Relocation problem. Two 16 KB programs loaded consecutively into memory.

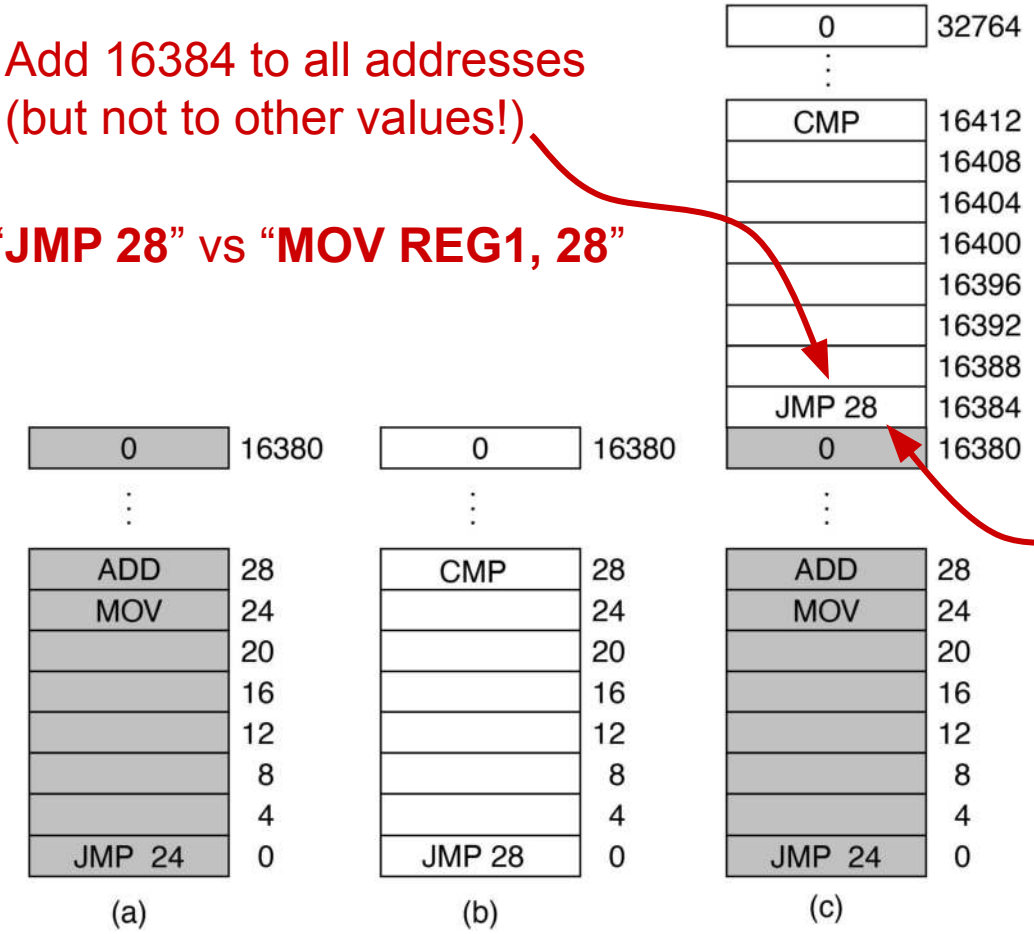
Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

No Memory Abstraction: Multiprogramming

Add 16384 to all addresses
(but not to other values!)

"**JMP 28**" vs "**MOV REG1, 28**"



Naive approach:

Move each program to a dedicated region

Problems:

- **Relocation**
  - Load-time?
- **Protection**
  - Memory keys?

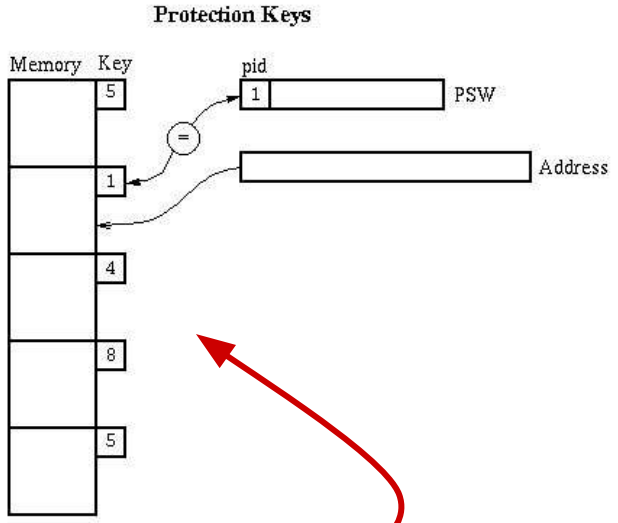


Figure 3-2. Relocation problem. Two 16 KB programs loaded consecutively into memory.

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Memory Abstraction: Address Spaces

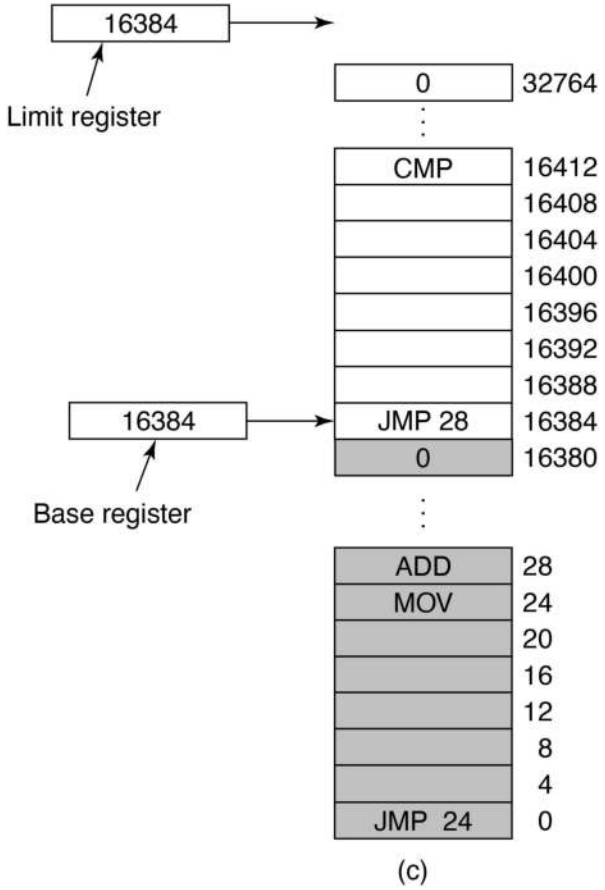


Figure 3-3. Base and limit registers

- Base register operates **dynamic relocation**
- Limit register enforces **protection**

Checks for MOV Reg1, Addr:

IF(Addr > LIMIT) → **NOT OKAY**

IF(BASE + Addr < BASE) → **NOT OKAY**

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

What if we run out of memory?

- Simplest solution: swapping of processes
  - bring in entire process, run it for a while, then put it back on disk/SSD
  - Idle processes do not take up any memory when they are not running

- Alternative: virtual memory ⇒ We will discuss this later!

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

An Improvement: Dynamic Partitions and Swapping (1/2)

Swapping may lead to memory fragmentation

→ *Memory compaction*

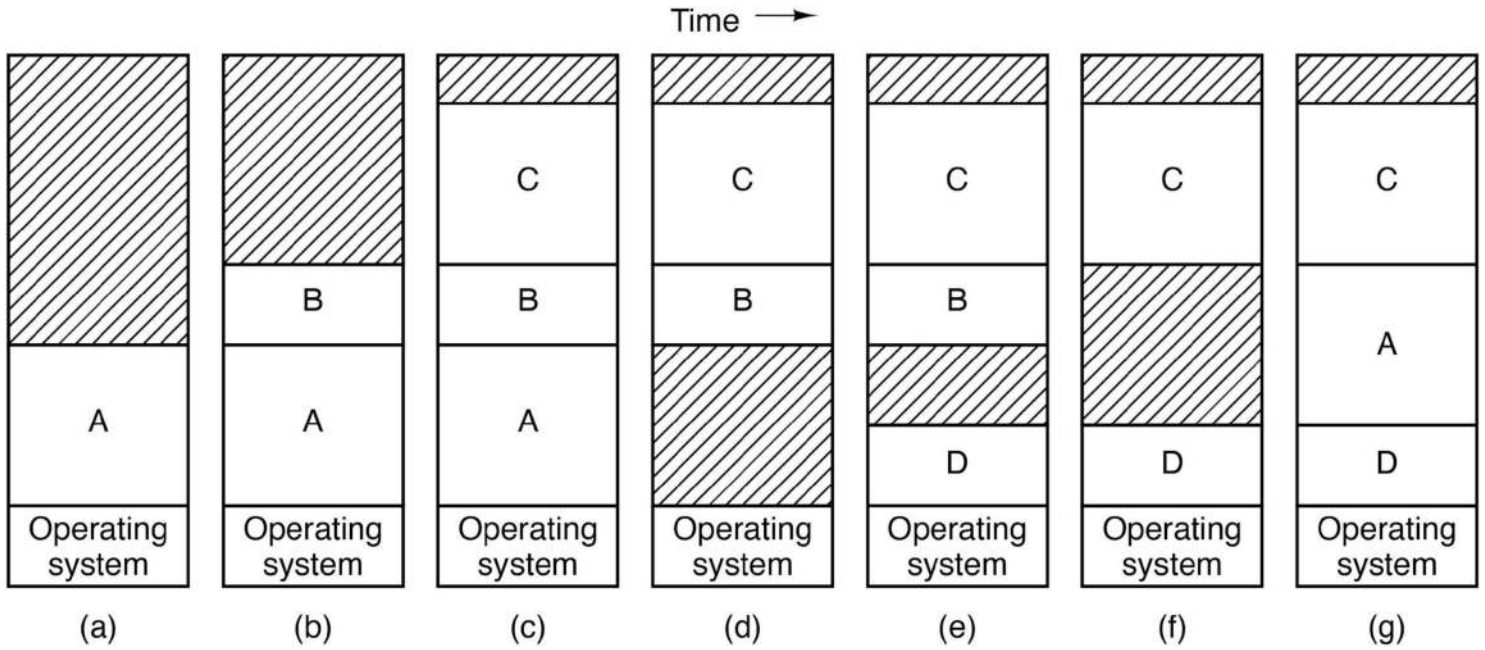


Figure 3-4. Memory allocation changes as processes come into memory and leave it.

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

An Improvement: Dynamic Partitions and Swapping (2/2)

Problem:

- We need to allow extra room for process growth

Issues:

- How much extra room?
  - Mem usage vs. OOM (Out-Of-Memory) risk
- What to do on OOMs?
  - Kill process
  - Relocate process
  - Swap out

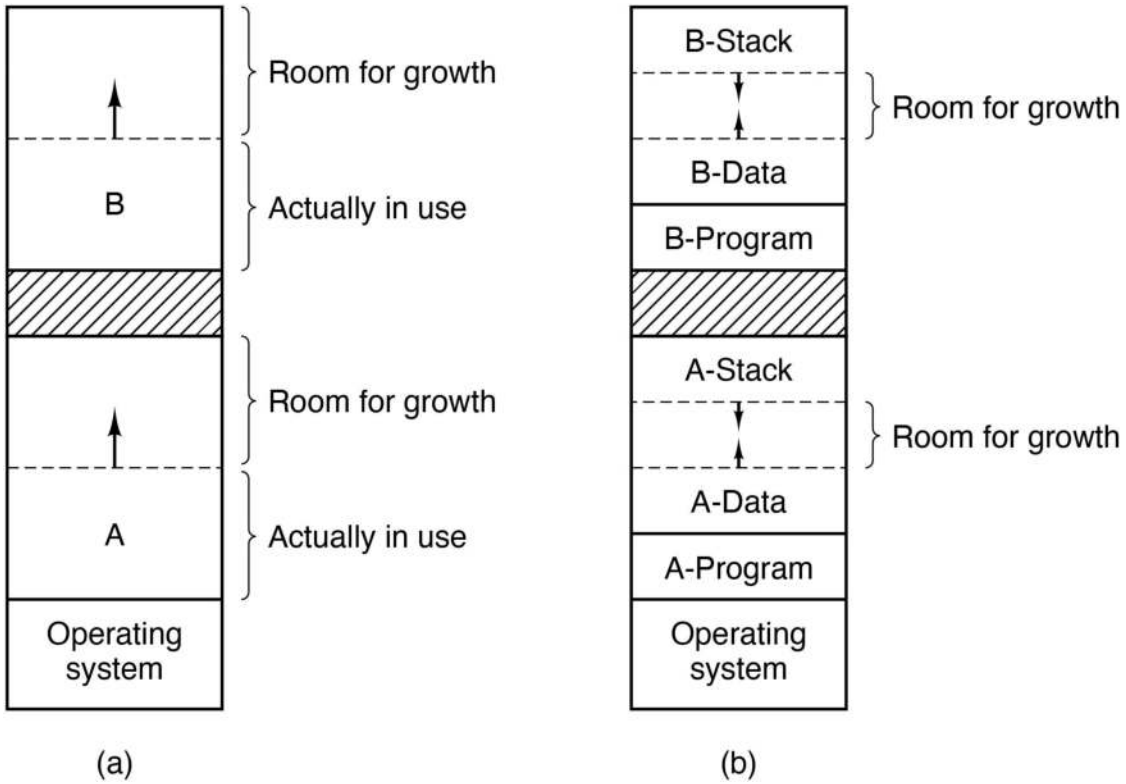


Figure 3-5. Allocating space for…
(a) … growing data segment
(b) … growing stack + growing data segment

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Memory Management

Question: Which part of the memory is allocated?

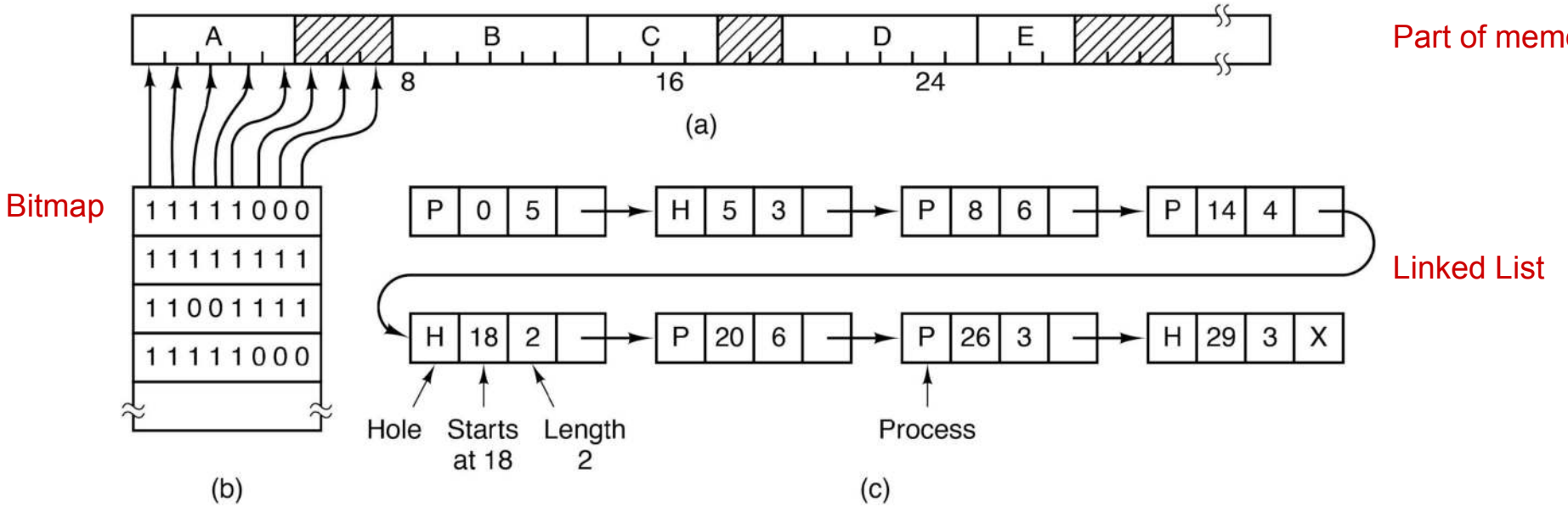
- Divide memory in blocks
  - e.g. each block is 4 bytes

- **Solution 1:** Bitmap keeps track of which blocks are allocated
- **Solution 2:** Linked list keeps track of unallocated memory

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Memory Management: Bitmaps vs Linked Lists



- **Bitmap**: finding holes requires (slow) scanning
- **List**: common vs. separate process/hole lists
  - Slow allocation vs. slow deallocation
  - Holes sorted by address for fast coalescing

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Memory Management with Linked Lists

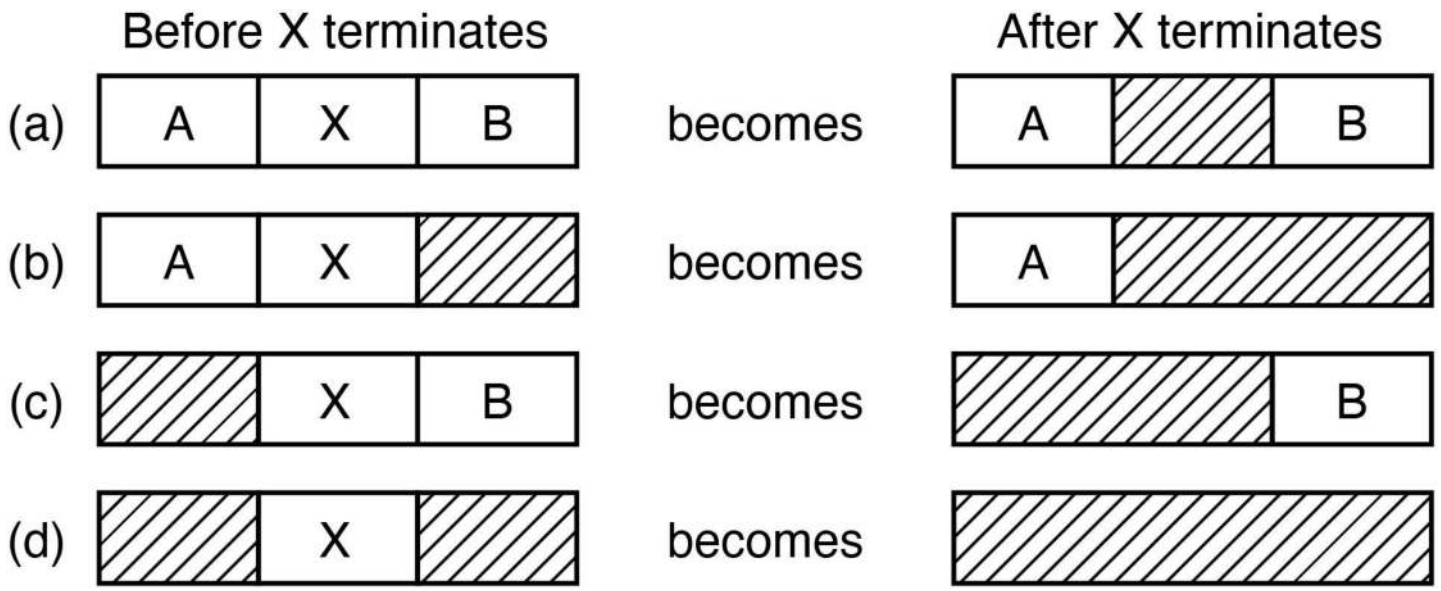


Figure 3-7. Four neighbor combinations for the terminating process, **X**.

In practice, a doubly linked is often used ⇒ makes freeing easy

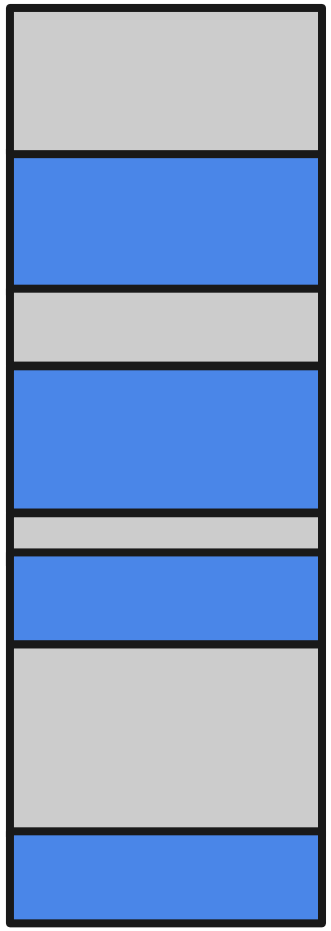
- Can easily check if previous free
- Can easily adjust the pointers

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Memory Allocation Schemes

- **First Fit**: Take first fitting hole (MINIX 3)
  - Simplest option
- **Next Fit**: Take next fitting hole
  - Slower than first fit in practice
- **Best Fit**: Take best fitting hole
  - Prone to fragmentation
- **Worst Fit**: Take worst fitting hole
  - Poor performance in practice
- **Quick Fit**: Keep holes of different sizes
  - Poor coalescing performance
- **Buddy allocation scheme** (Linux)
  - Improves quick fit's coalescing performance



Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Extra material

The next few slides contain information that is not found in Chapter 3. It illustrates how memory allocators work in practice,in this case, in an operating system such as Linux. You can find some of this information in Chapter 10.4.

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Buddy Memory Allocation

Extra material. See also: Sec.10.4

Originally: one area of size 64 “chunks”

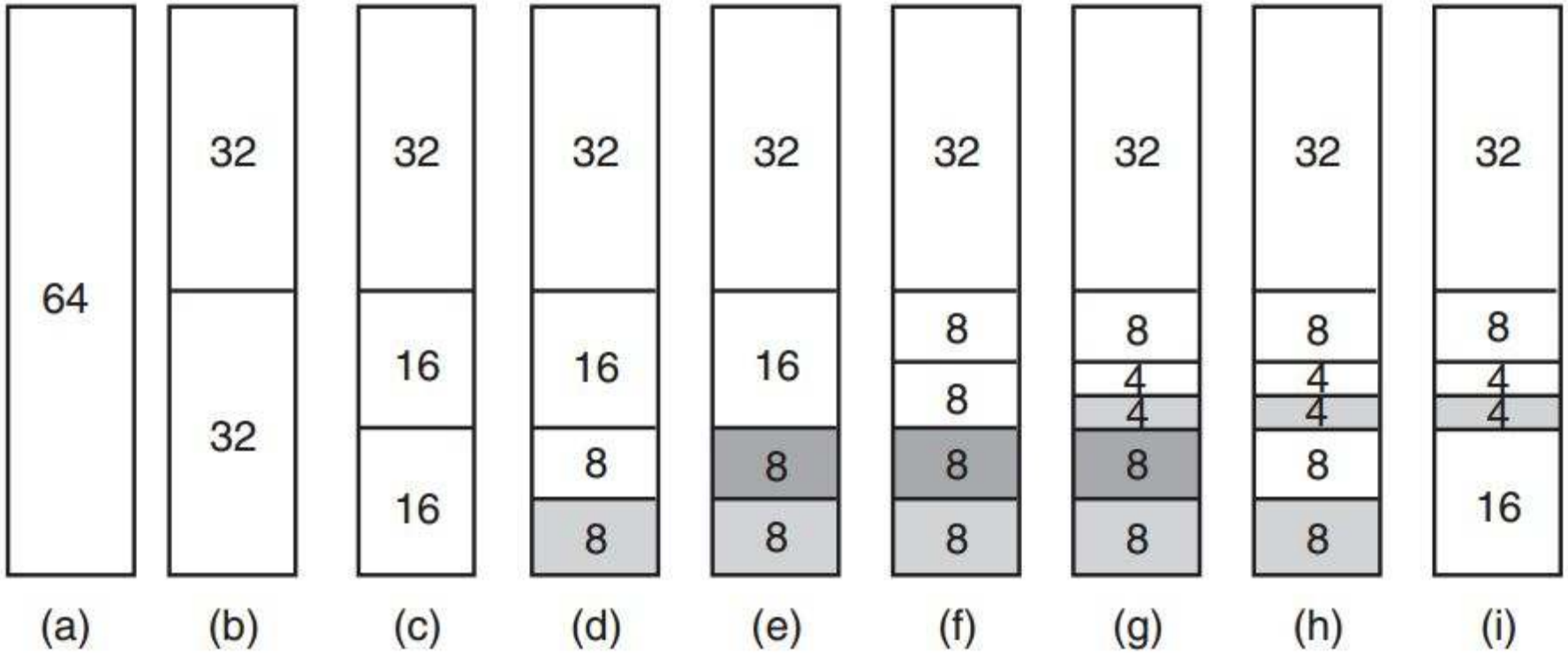


Figure 10-17. Operation of the buddy algorithm.

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Buddy Memory Allocation (Based on Wikipedia Example)

https://en.wikipedia.org/wiki/Buddy_memory_allocation

Step	4K	4K	4K	4K	4K	4K	4K	4K
2.1	2^3							
2.2	2^2				2^2			
2.3	2^1		2^1		2^2			
2.4	2^0	2^0	2^1		2^2			
2.5	A: 2^0	2^0	2^1		2^2			
3	A: 2^0	2^0	B: 2^1		2^2			
4	A: 2^0	C: 2^0	B: 2^1		2^2			
5.1	A: 2^0	C: 2^0	B: 2^1		2^1		2^1	
5.2	A: 2^0	C: 2^0	B: 2^1		D: 2^1		2^1	
6	A: 2^0	C: 2^0	2^1		D: 2^1		2^1	
7.1	A: 2^0	C: 2^0	2^1		2^1		2^1	
7.2	A: 2^0	C: 2^0	2^1		2^2			
8	2^0	C: 2^0	2^1		2^2			
9.1	2^0	2^0	2^1		2^2			
9.2	2^1		2^1		2^2			
9.3	2^2				2^2			
9.4	2^3							

Terminology:
Block of order *n*: (basic block size) * 2^n
Here, basic block size = 4K

The following requests may have occurred:

1. The initial situation.

2. A requests 4 K (⇒ order 0) → repeated splits (Steps 2.1-5)
3. B requests 9 K (⇒ order 1) → available (Step 3)
4. C requests 3 K (⇒ order 0) → available (Step 4)
5. D requests 7 K (⇒ order 1) → split order 2 (Steps 5.1-2)

6. B releases memory → free order 1 block (Step 6)
7. D releases memory → free order 1 + coalesce (Steps 7.1-2)
8. A releases memory → free order 0 block.(Step 8)
9. C releases memory → free order 0 block + 3 x coalesce

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Slabs and Buddies

Extra material

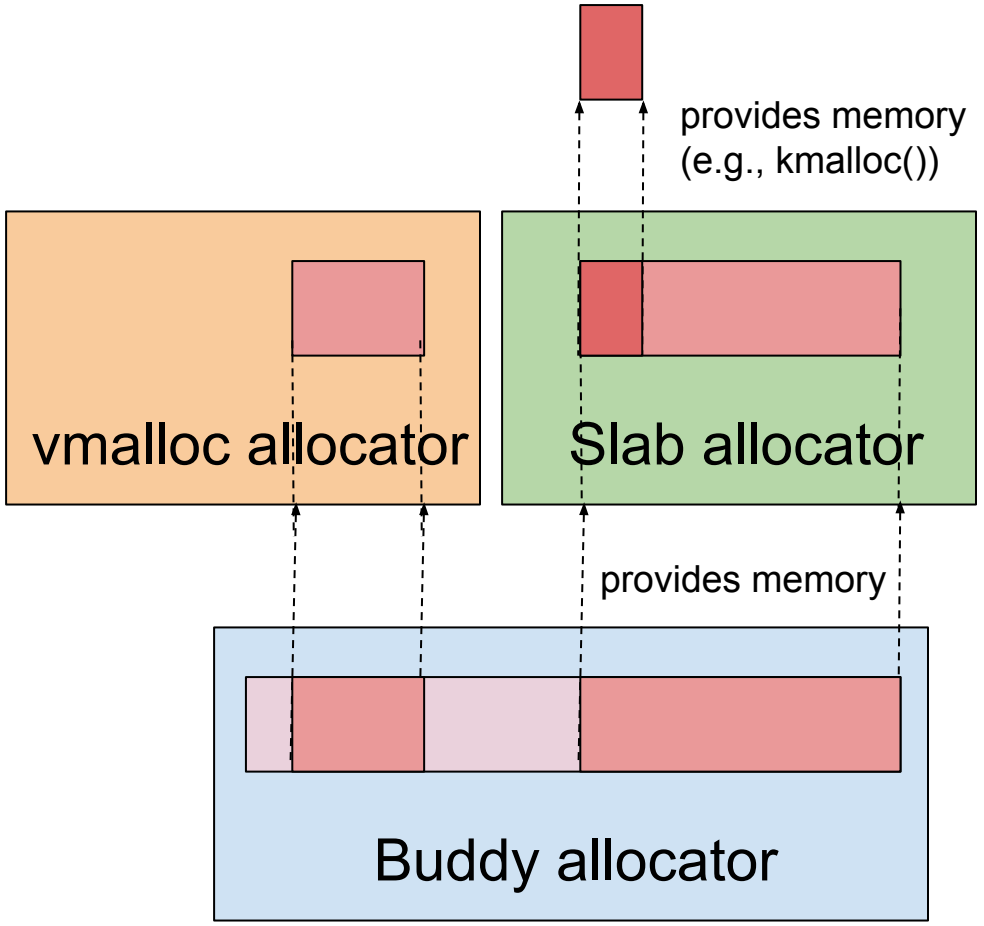
Buddy allocator:

- Fast and limits external fragmentation
- but considerable internal fragmentation

Often combined with other allocators

- E.g., in Linux, the vmalloc and slab allocators sit on top of buddy allocator

So slab allocator gets memory from buddy and parcels this out to callers of kmalloc()



Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Slab Allocator: Motivation

Extra material

Allocation + freeing of objects of the same type: very common in OS

- Speed is of the essence
- Unfortunately, allocating, initializing, and destroying objects all take time
  - E.g., list of free chunks → allocation requires traversing list to find a chunk of sufficient size

Goal:

- make metadata for allocator (e.g., to track what memory is free) very fast
- cache as much as possible

Solution:

- Slab allocator

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Slab Allocator: History

Extra material

Slab caches are an old idea: Jeff Bonwick (Sun Microsystems) 1994

Many incarnations

E.g., Linux:

- Adopted SLAB allocators in the 1990s.
- Multiple variants available in the kernel: SLOB, SLAB, SLUB

We will look at the general idea

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Slab Allocator: Caching makes things simple and fast

Extra material

To allocate an object:

```
if (there's an object in the cache) {
   take it (no construction required);
} else {
   allocate memory;
   construct the object;
}
```

To free an object:

```
return it to the cache (no destruction required);
```

To reclaim memory from the cache:

```
take some objects from the cache;
destroy the objects;
free the underlying memory;
```

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Slab Allocator: Caches for specific objects (or object sizes)

Extra material

Slab allocator supports mem allocation for kernel.

Kernel needs many different temporary objects:

- E.g., `dentry`, `mm_struct`, `inode`, `files_struct` structures.

Temporary kernel objects

- can be both small and large
- are often allocated and often freed (efficiency is crucial)

The buddy allocator, which operates with areas composed of entire frames of memory, is not suitable for this.

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Slab Allocator: Reuse and Lookups

Extra material

Goal: efficient memory allocation of objects

- Reduces fragmentation
- Retains allocated memory containing a data object of a certain type for reuse in later allocations of objects of the same type

By reusing freed objects, under favorable circumstances, you can avoid having to reinitialize them (leaving them partially filled in)

With only objects of the same type in a slab ⇒ fast lookup (simple index)

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Slab Allocator: Caches for specific objects (or object sizes)

Extra material

For instance: a slab cache for inodes or for 16B objects, etc.

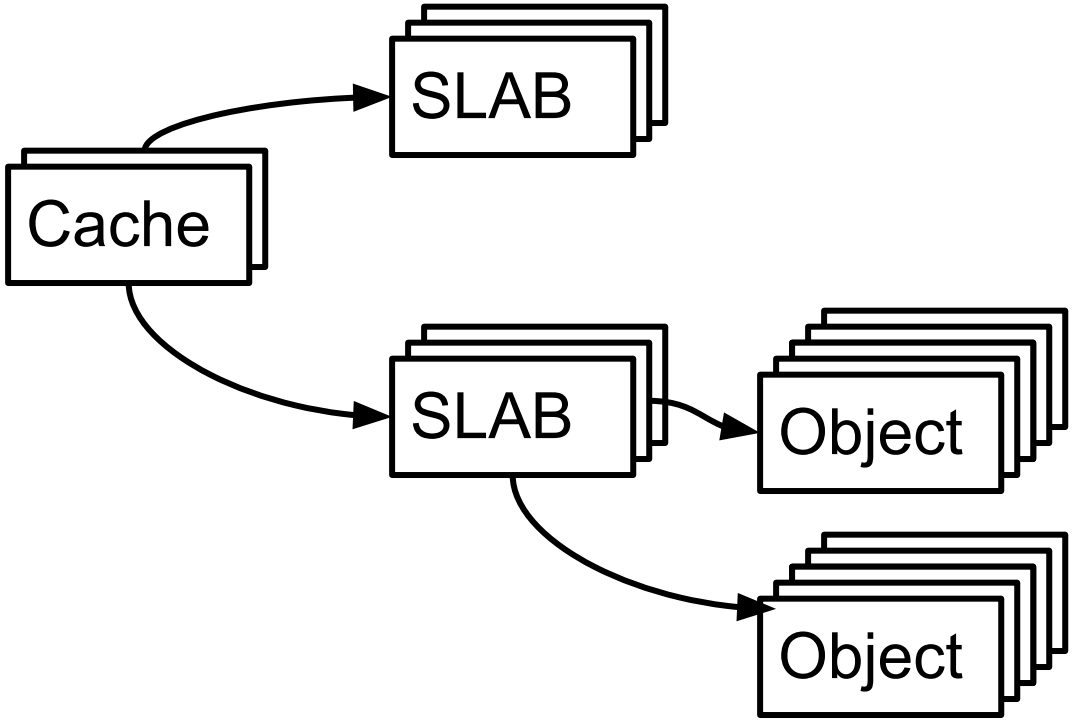
Slab primary unit of currency in slab allocator.

- When allocator needs to grow a cache: acquire an entire slab of objects.
- When allocator reclaims unused memory: relinquish a complete slab.

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Slab Cache: Organisation

Extra material



Each slab cache may contain multiple slabs, each of which contains (or points to) multiple objects

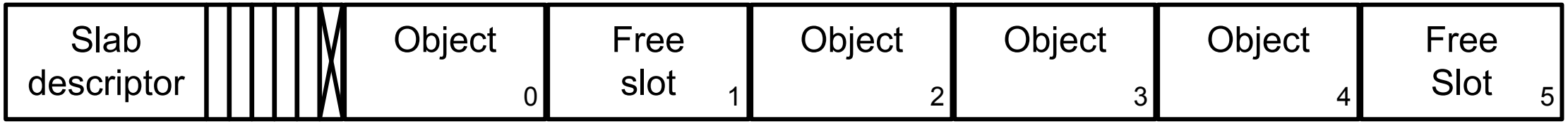
See: https://www.kernel.org/doc/gorman/html/understand/understand011.html#Sec:%20Tracking%20Free%20Objects

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Slab: Example (1/4)

Extra material



Note 1: Slabs can be implemented in different ways. We will look at one example

Note 2: All slots have the same (known) size, making it easy to find the n^{th} object in the slab

Pearson

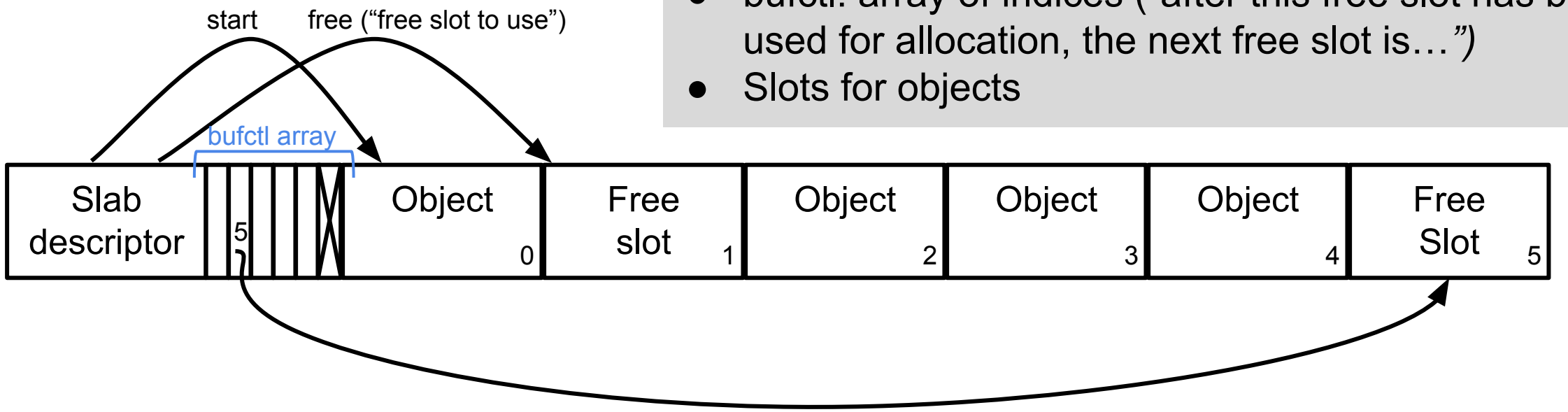
Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Slab: Example (2/4)

Extra material

In this example, a Slab contains:

- pointer to start of memory with object slots
- index of next free slot (“slot for next allocation”)
- bufctl: array of indices (“after this free slot has been used for allocation, the next free slot is…”)
- Slots for objects



Every entry in bufctl points to next free object (relative to this one). E.g: “the next free slot after this free slot (1) is the slot at index 5 (last slot)”

Initially: free =0, bufctl [0] = 1, bufctl [1]= 2, etc.
Allocation: free points to next free object

Deallocation of entry n:

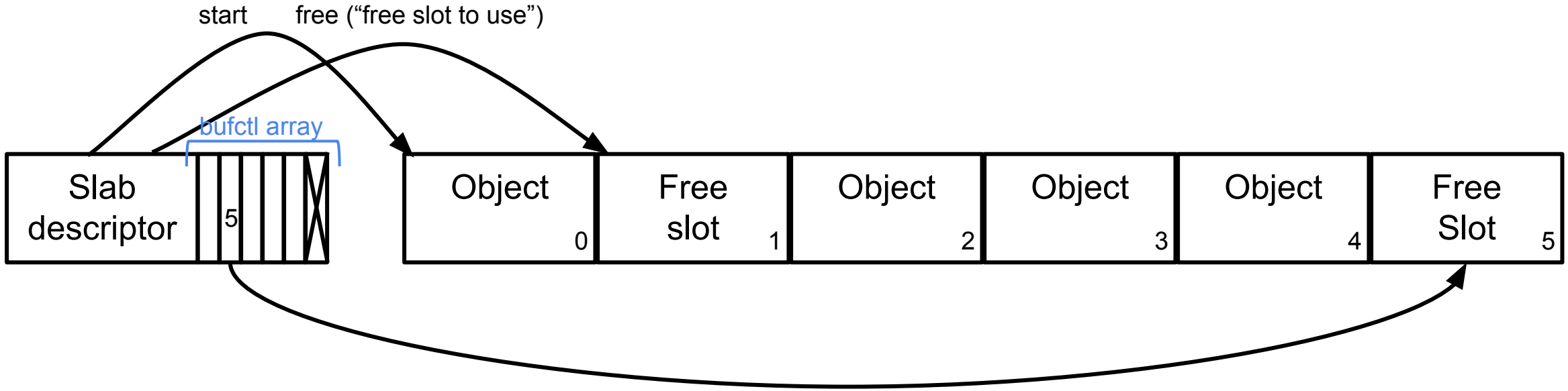
```
bufctl [n]:= free  ;
free := n:
```

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Slab: Example (3/4)

Extra material



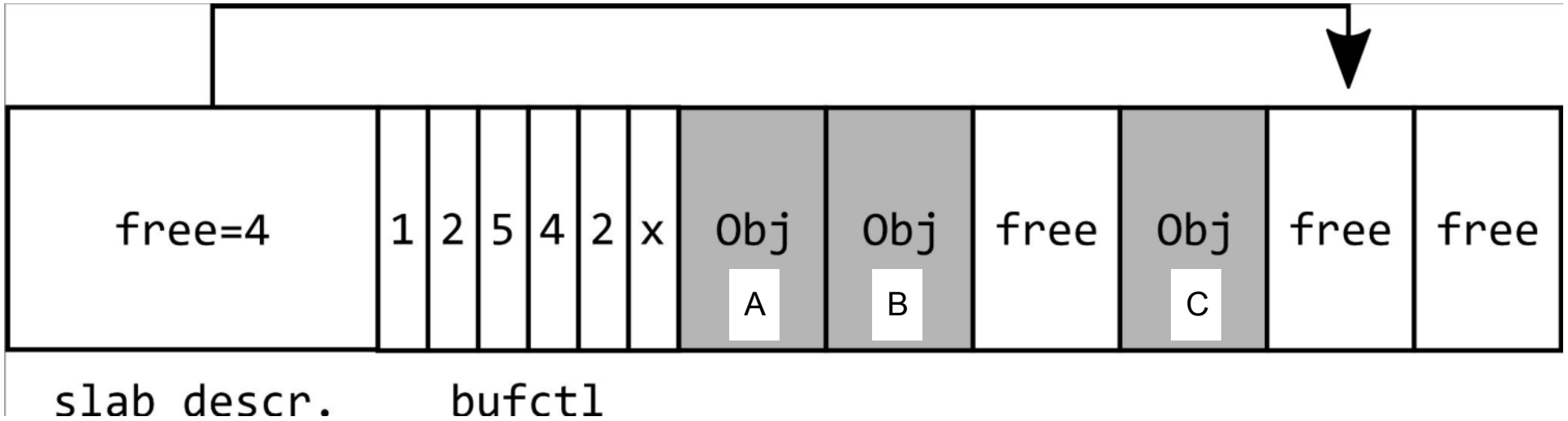
For larger objects the memory area can be decoupled from the descriptor

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Slab: Example (4/4) Quiz

Extra material



Given the initial state of the slab above and the following operations:

```
D = slab_alloc (obj);
free (B) ;
E = slab_alloc (obj);
F = slab_alloc (obj)
```

Q: Where is F allocated?

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

SLAB caches in Linux

Extra material

Dedicated ones and “general” ones.

General: different powers of two

The following dedicated caches exist:

```
extern struct kmem_cache     *vm_area_cachep;
extern struct kmem_cache     *names_cachep;
extern struct kmem_cache     *files_cachep;
extern struct kmem_cache     *fs_cachep;
extern struct kmem_cache     *sighand_cachep;
```

Check `/proc/slabinfo` for all slab caches

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

/proc/slabinfo

number of objects in use

total number of allocated objects

size of objects in this slab, in bytes

number of objects stored in each slab

number of pages allocated for each slab

number of active slabs

total number of slabs

Name	active objs	num objs	object size	objs per slab	pages per slab	active slabs	num slabs
inode_cache	9174	9660	568	28	4	345	345
dentry	1532244	1532244	192	42	2	36483	36483
buffer_head	794118	794118	104	39	1	20362	20362
vm_area_struct	195838	204286	176	46	2	4441	4441
mm_struct	7658	8352	896	36	8	232	232
files_cache	5831	7314	704	46	8	159	159
signal_cache	3111	3780	1088	30	8	126	126
sighand_cache	2094	2310	2112	15	8	154	154
task_struct	2793	3258	1776	18	8	181	181
anon_vma	91453	99584	64	64	1	1556	1556
radix_tree_node	360485	553224	568	28	4	19758	19758
dma-kmalloc-8192	0	0	8192	4	8	0	0
dma-kmalloc-4096	0	0	4096	8	8	0	0
dma-kmalloc-2048	0	0	2048	16	8	0	0
dma-kmalloc-1024	0	0	1024	32	8	0	0
dma-kmalloc-512	32	32	512	32	4	1	1
dma-kmalloc-256	0	0	256	32	2	0	0
dma-kmalloc-128	0	0	128	32	1	0	0
dma-kmalloc-64	0	0	64	64	1	0	0
dma-kmalloc-32	0	0	32	128	1	0	0
dma-kmalloc-16	0	0	16	256	1	0	0
dma-kmalloc-8	0	0	8	512	1	0	0
dma-kmalloc-192	0	0	192	42	2	0	0
dma-kmalloc-96	0	0	96	42	1	0	0
kmalloc-8192	256	268	8192	4	8	67	67
kmalloc-4096	1595	1680	4096	8	8	210	210
kmalloc-2048	3200	3664	2048	16	8	229	229
kmalloc-1024	6786	7584	1024	32	8	237	237
kmalloc-512	12203	36096	512	32	4	1128	1128
kmalloc-256	64100	101408	256	32	2	3169	3169
kmalloc-128	138762	215136	128	32	1	6723	6723
kmalloc-64	482131	782976	64	64	1	12234	12234
kmalloc-32	25499	33024	32	128	1	258	258

End of extra material

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

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
  - Initially, only the overlay manager (OM) was loaded
  - The OM would then load overlay 0 and run it
  - When done, it would tell the OM to load overlay 1, etc.
- Nowadays, we have a better solution: *virtual memory*
  - Often implemented using *paging*

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Virtual memory

Problem: So far memory can only be given to processes in contiguous pieces

Solution:

- Create for the process the illusion of a large (e.g., 48 bit) address space
- This is known as the *virtual* address space
- The (much more limited) RAM is known as *physical* memory
- Translate virtual addresses (as used by the process) into physical addresses
- Typically done in the MMU

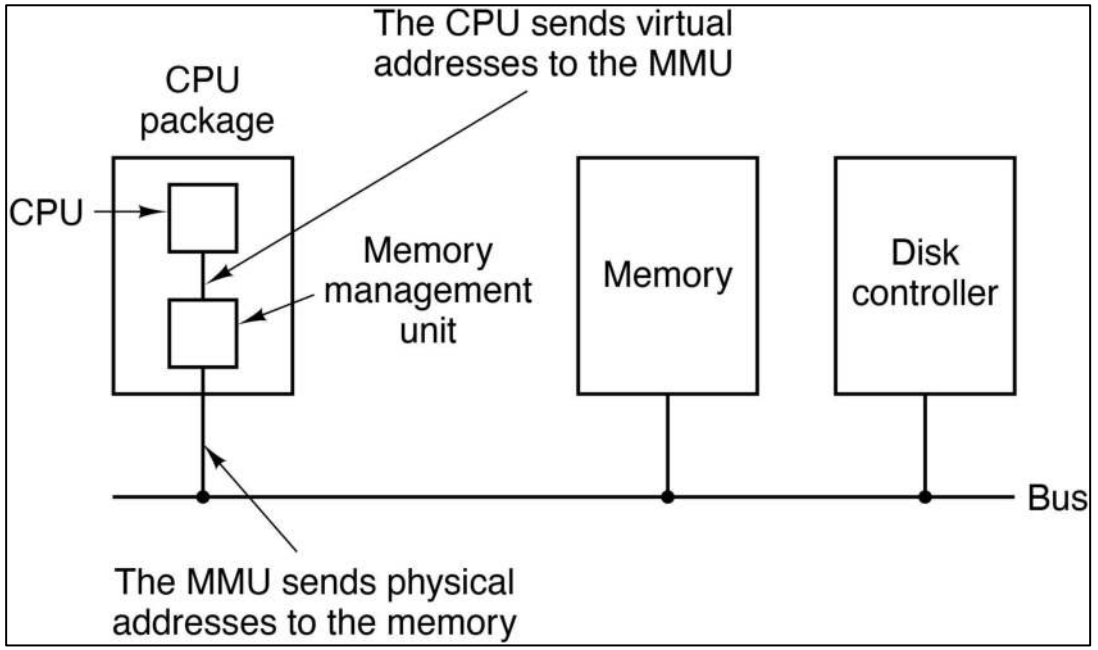


Figure 3-8. The position and function of the MMU.

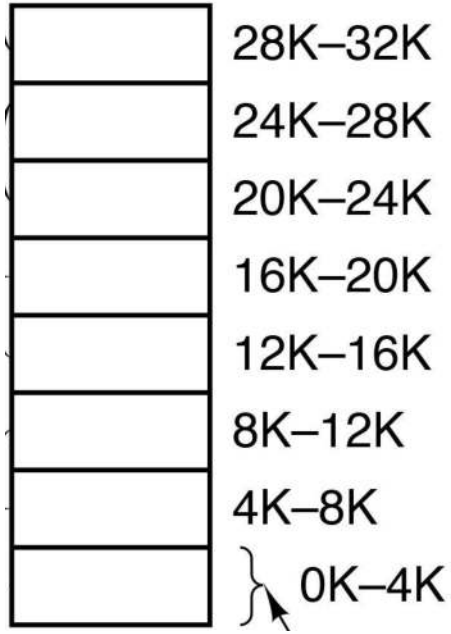
Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Virtual memory and paging

Modern systems use paging:

- Divide physical and virtual memory into pages of fixed size (e.g. 4096 bytes)
- Translate virtual pages into physical pages (frames)



Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Virtual Address Space vs. Physical Address Space and Page Tables

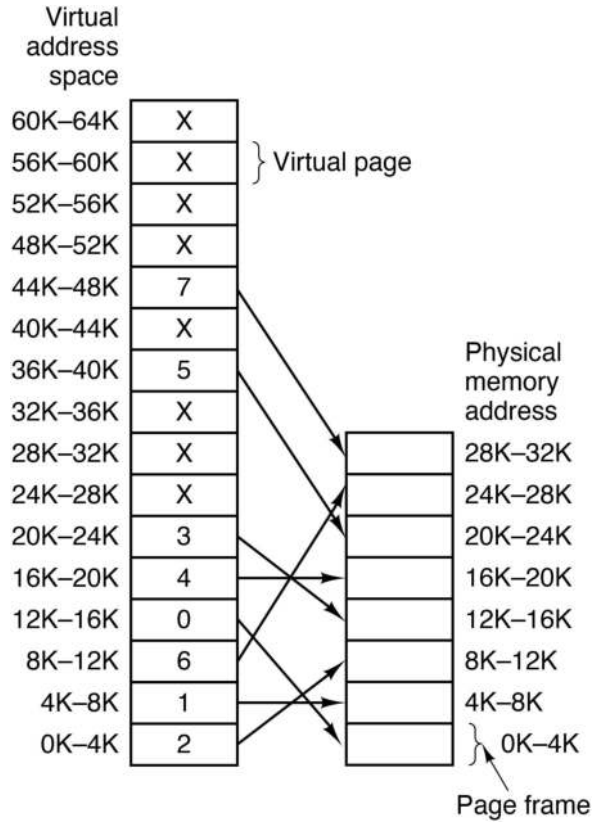


Figure 3-9. The relation between virtual and physical memory addresses is given by the page table.

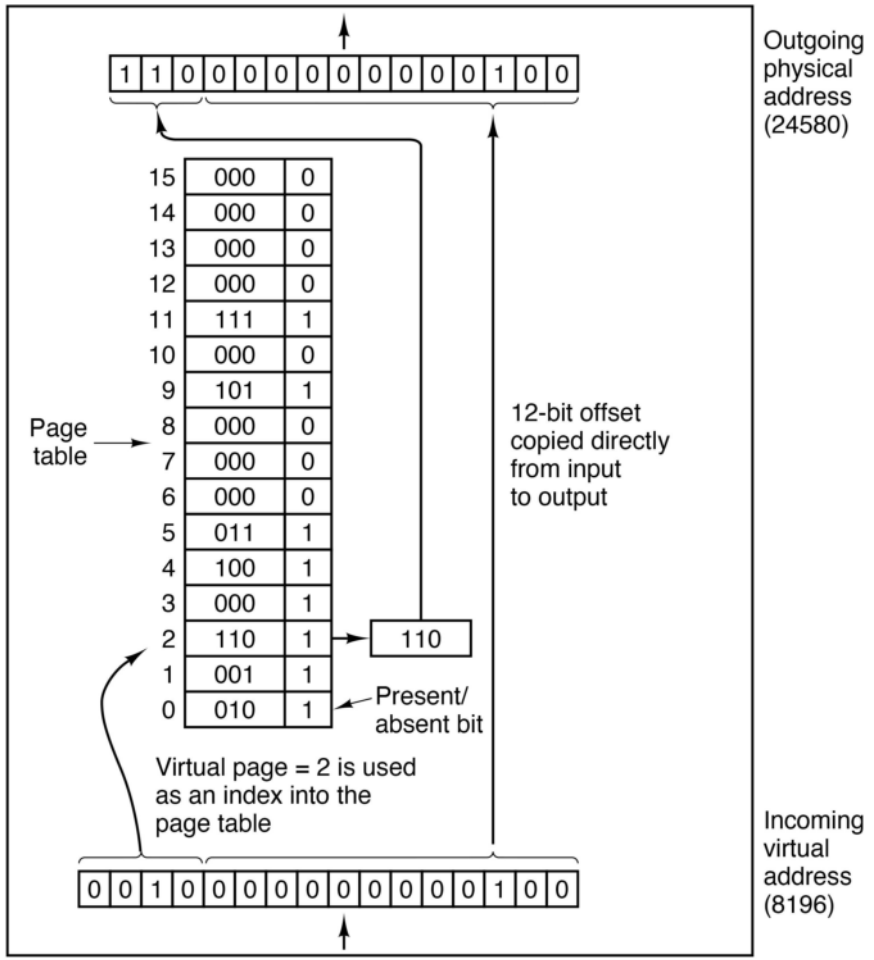


Figure 3-10. The internal operation of the MMU with 16 4-KB pages.

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Structure of a Page Table Entry

Questions:

- How big is the page table?
- How many page tables?

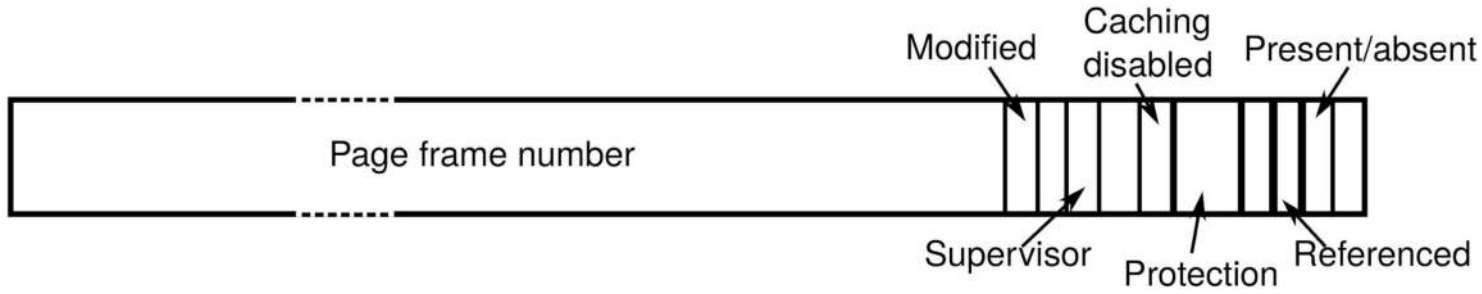


Figure 3-11. A typical page table entry.

- 4KB page size (typical)
- 32 bit systems:
  - 32 bit address space and 32 page table entries (PTEs)
- 64 bit systems
  - really: 48-bit or 57-bit address space (!) and 64-bit PTEs
  -

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Page Table Sizes

A very large virtual address space would lead to a very large page table → not nice!

Possible solutions:

- multi-level page tables
- inverted page tables

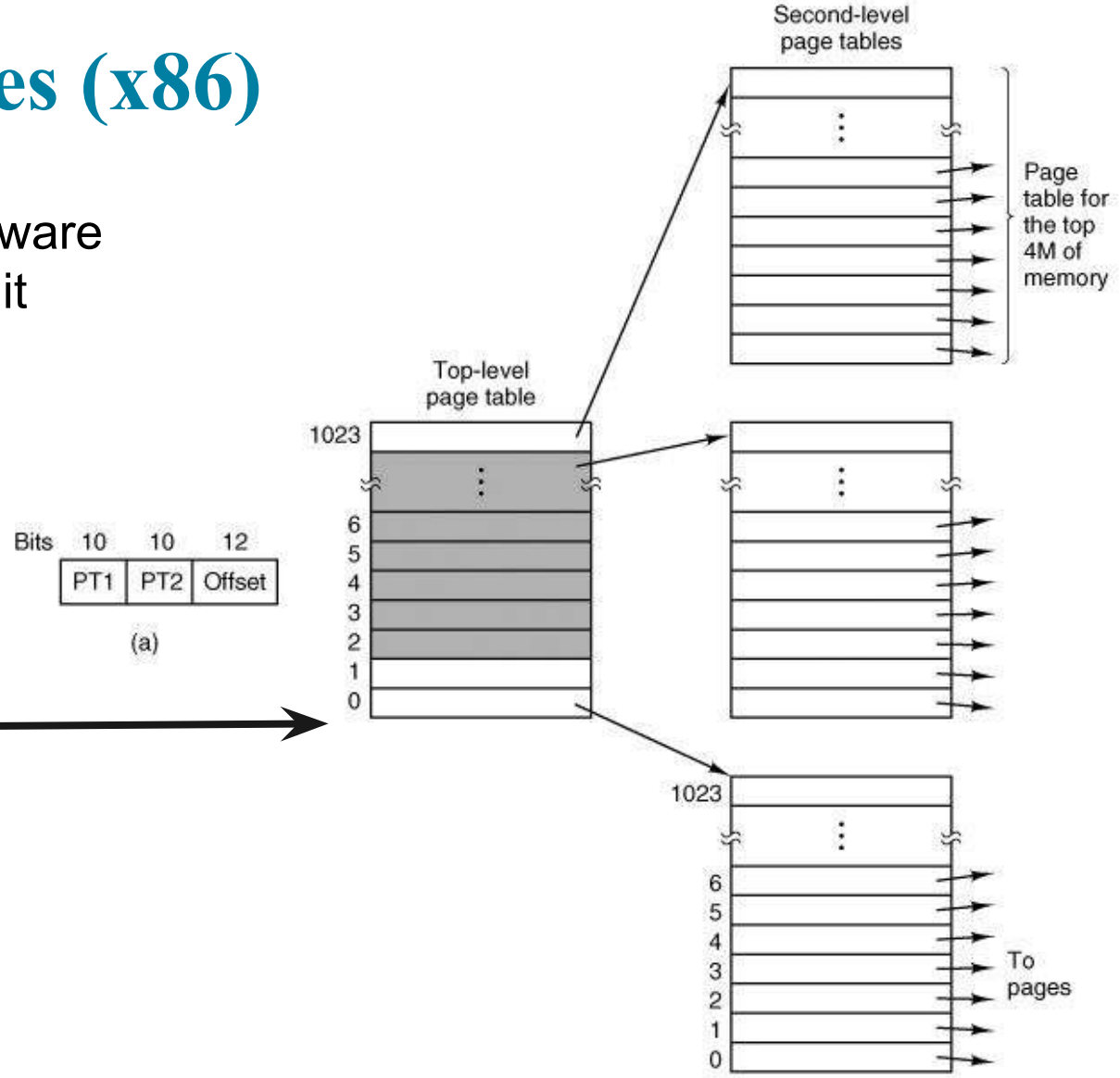
Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Two-level Page Tables (x86)

Page tables are “walked” by hardware
MMU = Memory Management Unit

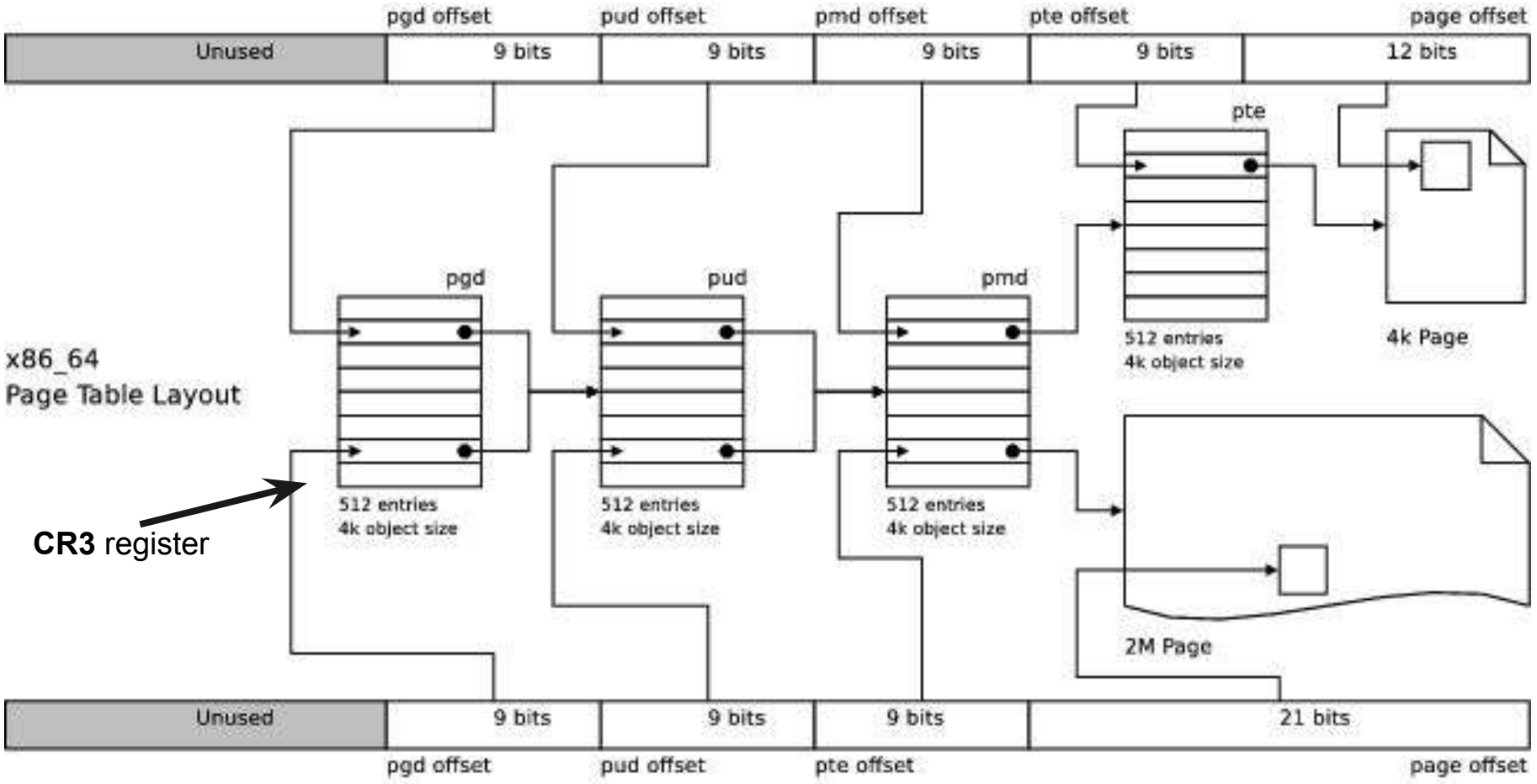
CR3 register
(special register to point to
top of page table hierarchy)



Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Four-level Page Tables (x86-64)



PGD: Page Global Directory, **PUD**: Page Upper Directory, **PMD**: Page Mid-level Directory, **PTE**: page table entry

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Inverted Page Tables (IA-64)

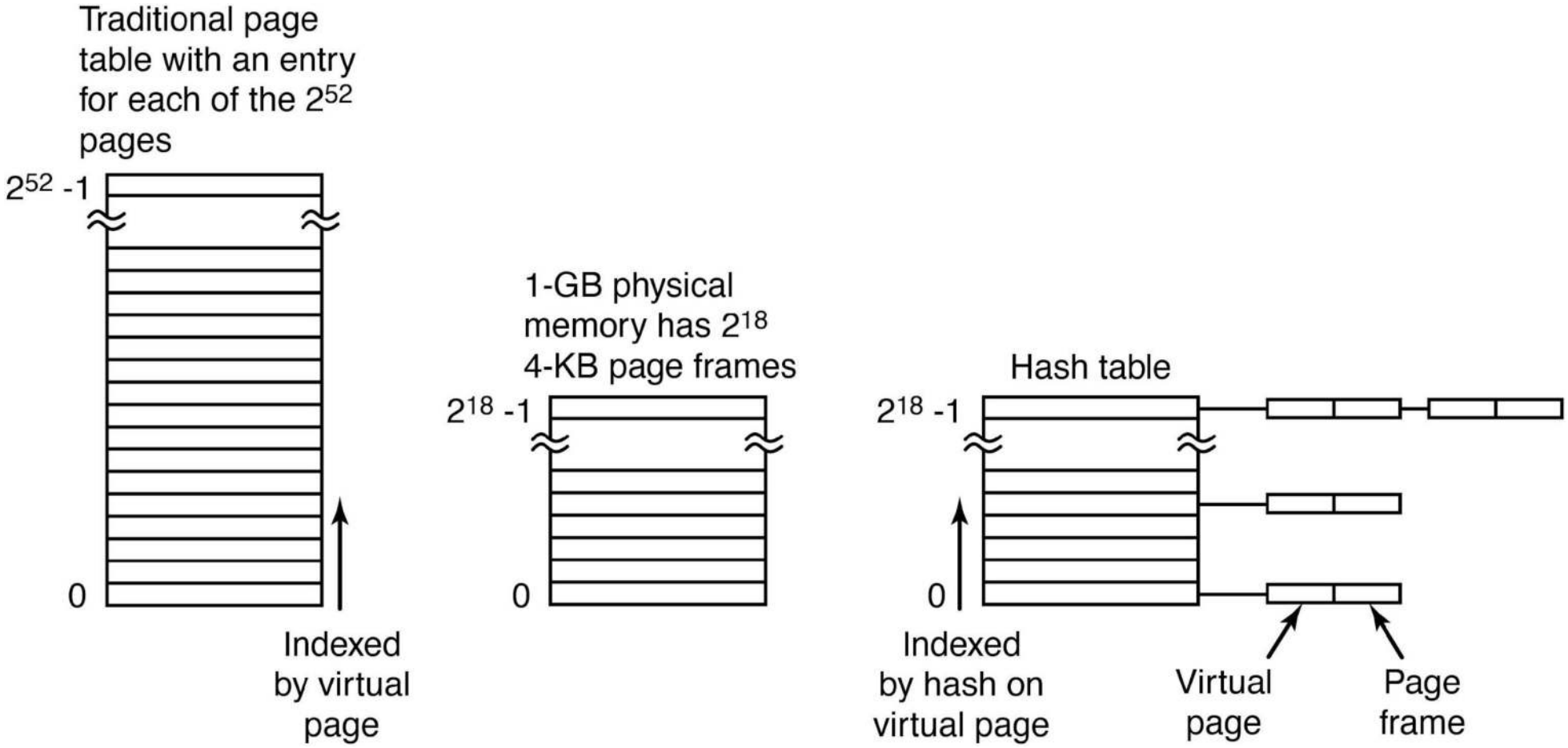


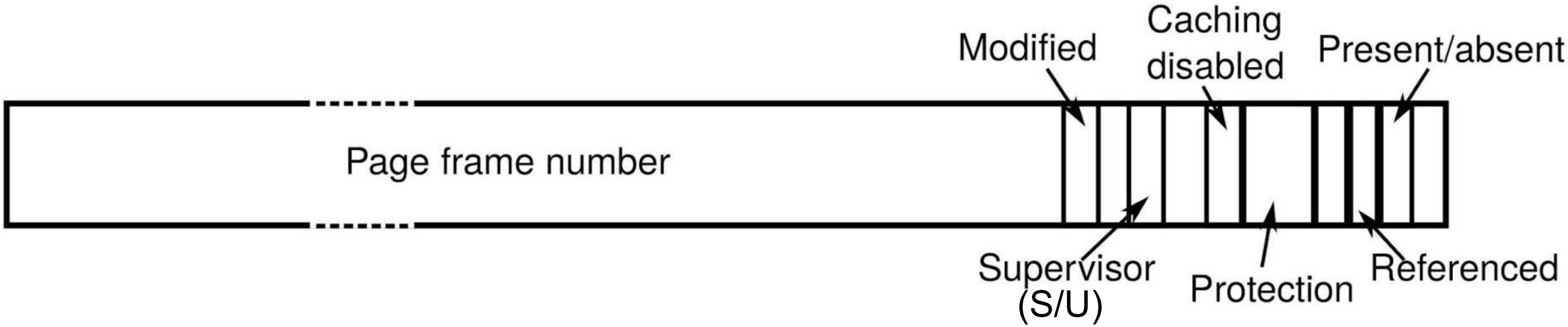
Figure 3-14. Comparison of a traditional page table with an inverted page table.

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Going back to PTEs a minute…

Extra material



We have these at each level

What happens if we combine them?

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

What happens if…

Extra material

1. PD entry says : `S/U==U, Prot==RO` (*)
   PT entry says : `S/U==U, Prot==R/W`

2. PD entry says : `S/U==U, Prot==RW`
   PT entry says : `S/U==U, Prot==RO`

3. PD entry says : `S/U==U`
   PT entry says : `S/U==S`

4. PD entry says : `S/U==S`
   PT entry says : `S/U==U`

*S/U simply means supervisor flag (which can be **S**upervisor or **U**ser)

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Combining Protection of Both Levels of Page Tables

Extra material

From the Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A:

> *For any one page, the protection attributes of its page-directory entry (first-level page table) may differ from those of its page-table entry (second-level page table). The processor checks the protection for a page in both its page-directory and the page-table entries. Table 4-3 shows the protection provided by the possible combinations of protection attributes when the WP flag is clear.*

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Extra material

Perhaps surprising?

Note: CR0 is a special control register in x86

- In principle: protection = strictest subset
- However, if CR0.WP=0, supervisor always has RW access

Table 4-3. Combined Page-Directory and Page-Table Protection

Page-Directory Entry		Page-Table Entry		Combined Effect	
Privilege	Access Type	Privilege	Access Type	Privilege	Access Type
User	Read-Only	User	Read-Only	User	Read-Only
User	Read-Only	User	Read-Write	User	Read-Only
User	Read-Write	User	Read-Only	User	Read-Only
User	Read-Write	User	Read-Write	User	Read/Write
User	Read-Only	Supervisor	Read-Only	Supervisor	Read/Write*
User	Read-Only	Supervisor	Read-Write	Supervisor	Read/Write*
User	Read-Write	Supervisor	Read-Only	Supervisor	Read/Write*
User	Read-Write	Supervisor	Read-Write	Supervisor	Read/Write
Supervisor	Read-Only	User	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Only	User	Read-Write	Supervisor	Read/Write*
Supervisor	Read-Write	User	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Write	User	Read-Write	Supervisor	Read/Write
Supervisor	Read-Only	Supervisor	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Only	Supervisor	Read-Write	Supervisor	Read/Write*
Supervisor	Read-Write	Supervisor	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Write	Supervisor	Read-Write	Supervisor	Read/Write

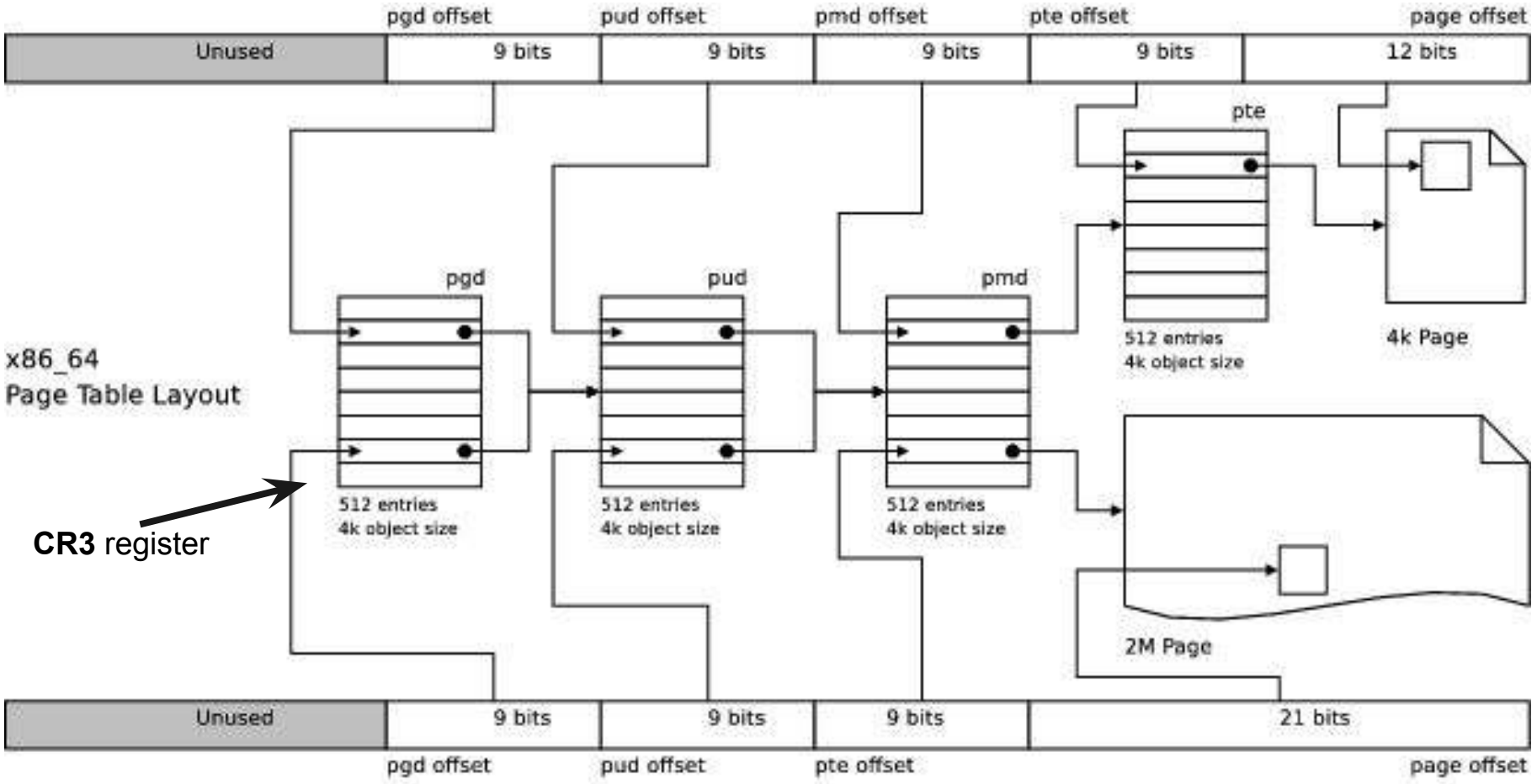
NOTE:

* If CR0.WP = 1, access type is determined by the R/W flags of the page-directory and page-table entries. IF CR0.WP = 0, supervisor privilege permits read-write access.

Pearson

Four-level Page Tables: Lots of Memory Accesses!

How many memory accesses for reading a single byte?



PGD: Page Global Directory, **PUD**: Page Upper Directory, **PMD**: Page Mid-level Directory, **PTE**: page table entry

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Translation Lookaside Buffer

Problem: MMU has to translate every single memory access → Performance problem

Solution: Cache previous translations, hoping programs exhibit a high degree of locality.

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.

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Translation Lookaside Buffer

Extra material

Questions:

- How many TLBs?
- When to flush TLB?
- When to expect many TLB misses?

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

Pearson

Figure 3-12.

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Translation Lookaside Buffer

Extra material

Questions:

- How many TLBs?
- When to flush TLB?
- When to expect many TLB misses?

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

1. Super expensive memory: small number of entries (e.g., 64-128)
2. Sometimes separate entries for huge (2MB) pages and normal (4KB) pages
3. Often separate TLBs for code and data
4. 1 per CPU, but sometimes multiple levels of TLB (L1 TLB vs L2 TLB)

Figure 3-12.

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Translation Lookaside Buffer

Extra material

Questions:

- How many TLBs?
- When to flush TLB?
- When to expect many TLB misses?

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

1. When entries change
2. Traditionally: at context switch?
3. Improvement: tag TLB entries with id of process (no need to flush if no other process uses your id)

Pearson

Figure 3-12.

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Translation Lookaside Buffer

Extra material

Questions:

- How many TLBs?
- When to flush TLB?
- When to expect many TLB misses?

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

1. After flushes
2. When the program exhibits little locality

Pearson

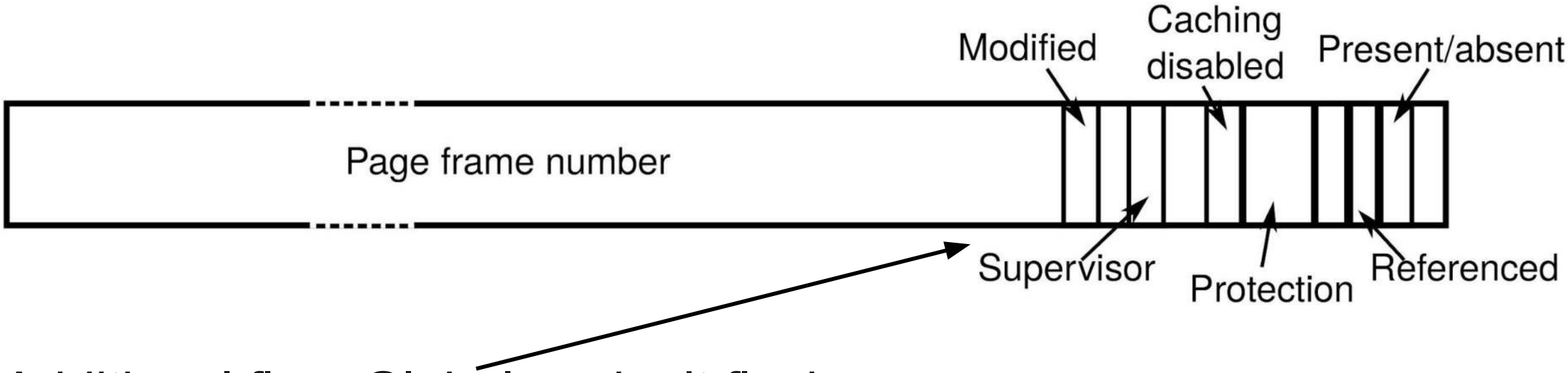
Figure 3-12.

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Translation Lookaside Buffer

Extra material

BTW **when not to flush?**



Additional flag: **G**lobal ⇒ don't flush

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

TLB miss and PF Handling

Software- vs. hardware-managed TLB:

- TLB misses handled by OS vs. hardware → Hardware walks PTs and fills TLB
- Flexibility vs. Efficiency

  ↓

  E.g., OS may preload TLB entries that it expects to need later (say the entries for a server to which current process sends msg)

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

TLB Misses and Page Fault Handling

- TLB miss handling:
  - Walk page tables to find the mapping
    - If mapping found, fill new TLB entry (**soft miss**)
    - If mapping not found, page fault (**hard miss**)

- OS handles PFs (similar to interrupts), cases:
  - Access violation (**segmentation fault**)
  - Legal access, PF handler needs to fix up page tables:
    - The page is already in memory (**minor PF**) OR
    - The page must be fetched from disk (**major PF**)

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Page Replacement

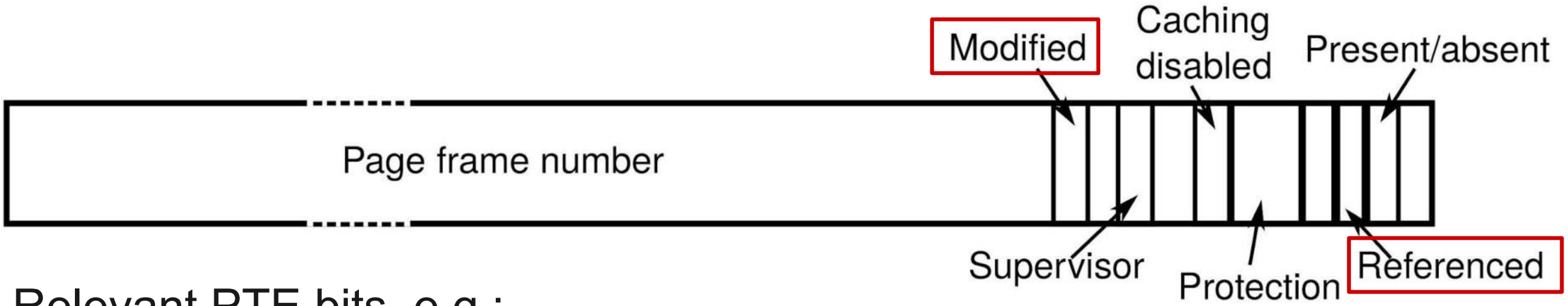
- Computer might use more virtual memory than it has physical memory
- Paging creates the illusion of unlimited memory available to user processes
- When a logical page is not in memory (swapped out to a file/partition), the OS has to page it in on a page fault
- Swapping out: direct or indirect reclaim
- Another logical page has to be swapped out ⇒ *which?*

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Page Replacement: Hardware Support

Extra material



Relevant PTE bits, e.g.:

- Modified (**M**): Set when page modified (aka “dirty” bit)
- Referenced (**R**): Set when page accessed (aka “accessed” bit)

- Available: Reserved for SW extensions
  - e.g., Soft-dirty bit on Linux, set by the kernel at every write page fault (PF) → because “dirty” bit is also used for other internal purposes

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

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

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Page Replacement: Basic Algorithms

- **Optimal:**
  - Replace page that will be referenced as far in the future as possible
  - Can this algorithm be implemented in practice?
- **Random:**
  - Simply replace a page at random
- **Not Recently Used (NRU)**:
  - Classes of pages: **Class 0:** R=0, M=0, **Class 1:** R=0, M=1, **Class 2:** R=1, M=0, **Class 3:** R=1, M=1
  - Periodically clear **R/M** bit for all the pages. Replace a page at random, but prioritize class 0, then class 1, etc.
- **FIFO**:
  - Build queue of faulting pages and replace the head

However, oldest page may still be useful

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Page Replacement: Second Chance

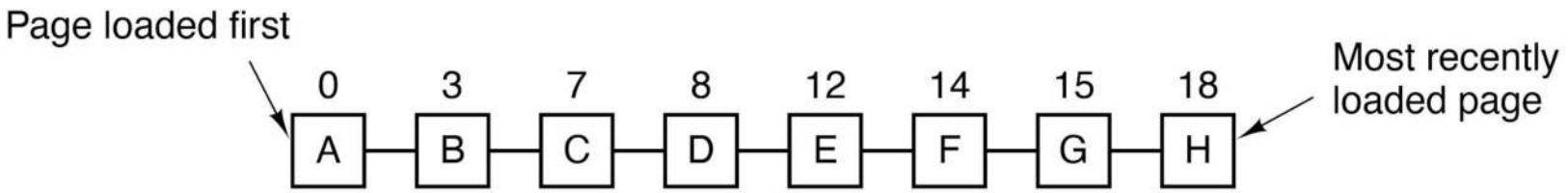


Figure 3-15 (a)

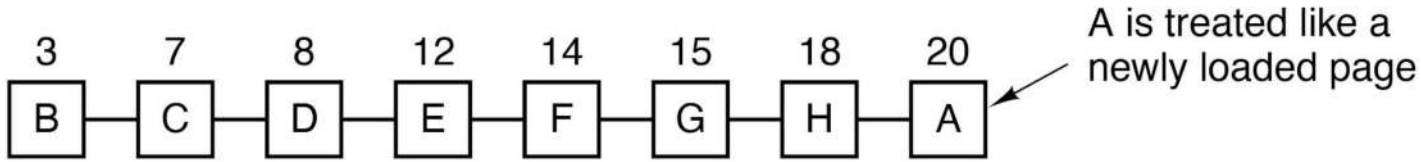


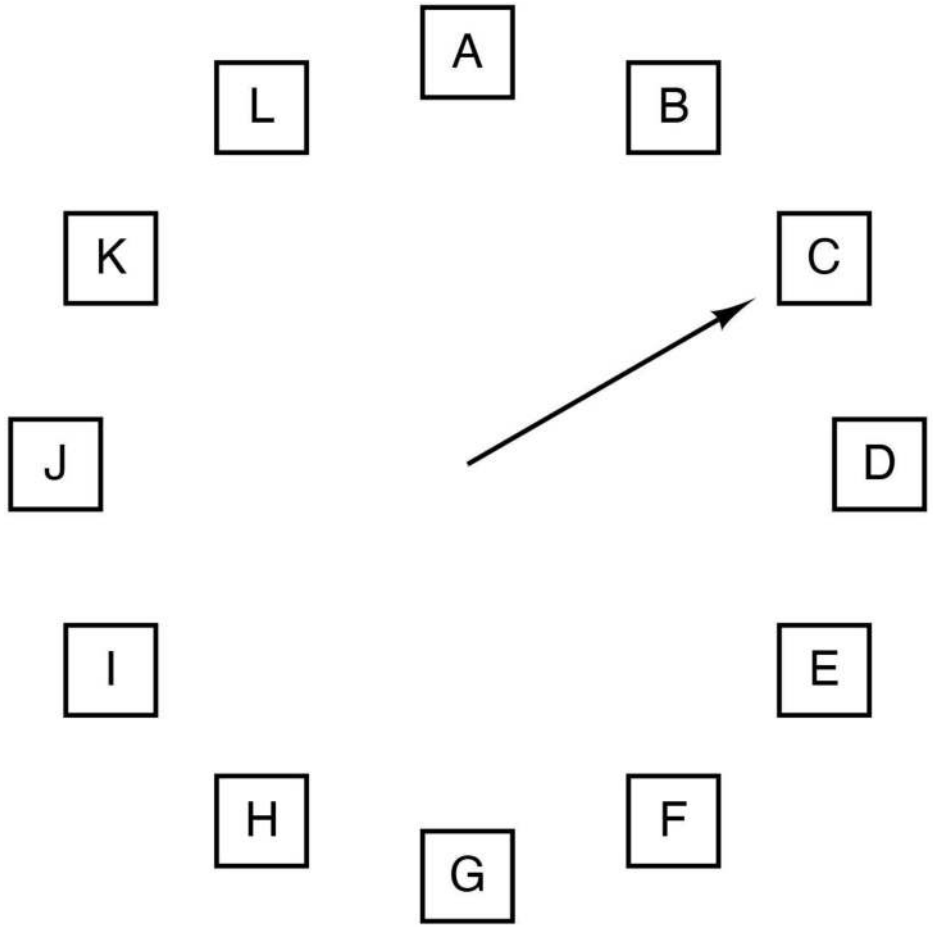
Figure 3-15 (b)

- Improved FIFO to preserve important pages
- For each visited page:
  - If **R**=0 then replace page
  - If **R**=1 then set **R**=0 and move page to tail

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Page Replacement: CLOCK



Hand points to oldest page

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

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Page Replacement: Least Recently Used (LRU)

- **Idea:** Replace page that has been unused for the longest
  - Good predictor for optimal in many (but not all) cases
  - Can LRU ever perform worse than random?
- **HW-assisted solutions:**
  - Maintain an ordered page list at each reference
    - Head is LRU page
  - Maintain reference timestamps in PTEs
    - Oldest-timestamp PTE points to LRU page
  - Rarely implemented in practice

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Page Replacement: LRU-like Strategies

- **Not Frequently Used (NFU):**
  - Maintain per-page counters and periodically add (and clear) the **R** bit to every counter
  - Replace the page with the lowest counter
  - Problem: NFU never “forgets” pages
- **Aging**:
  - Improve NFU by:
    - Add **R** bit to the leftmost bit
    - Shifting counters to the right
  - Offers lower temporal resolution than “real” LRU
  - **CLOCK** (and variations) more often used in practice

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Simulating LRU in Software

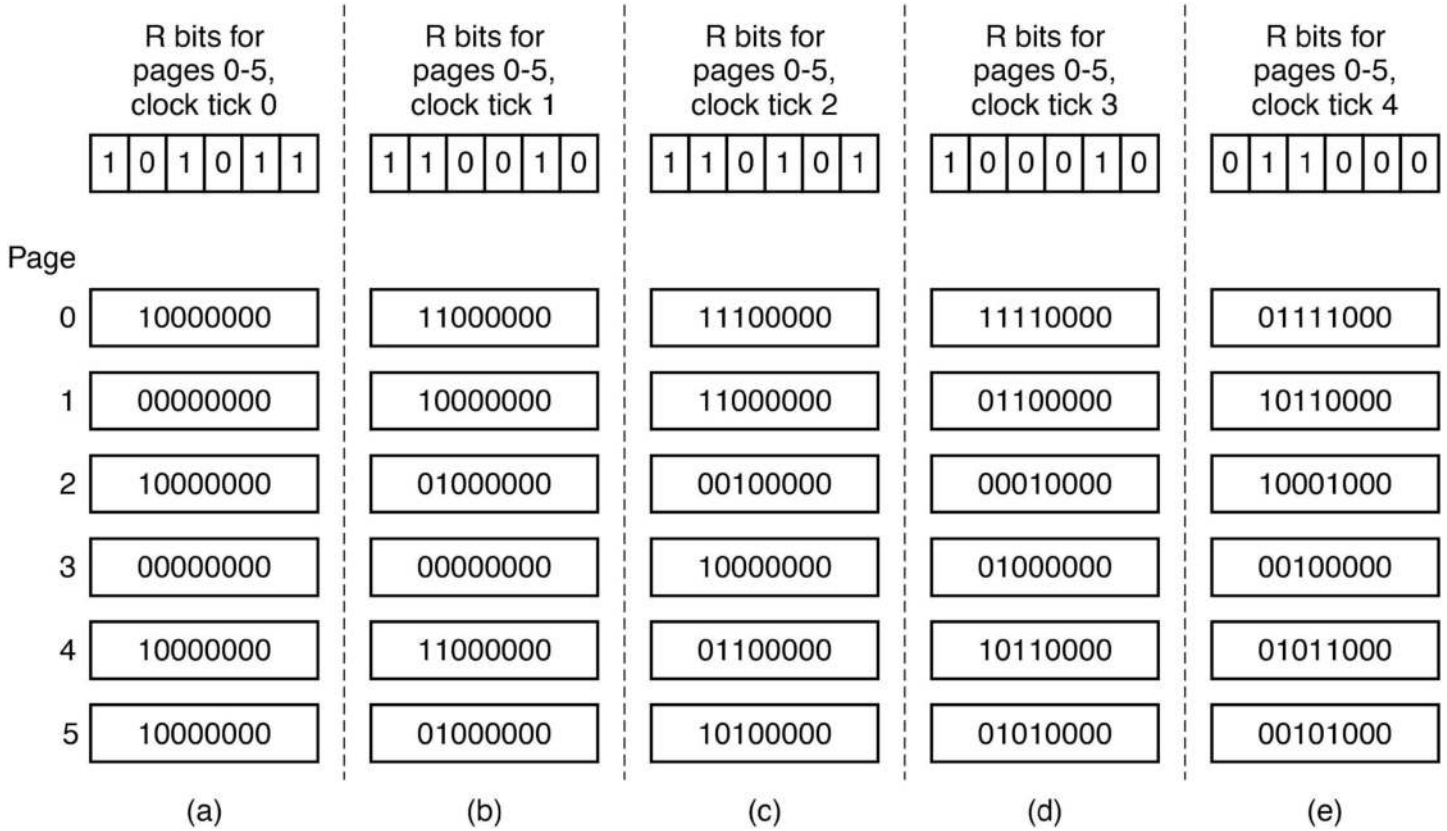


Figure 3-17. The aging algorithm simulates LRU in software.
Shown are six pages for five clock ticks. The five clock ticks are represented by (a) to (e).

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Page Replacement: WS and WSClock

Working Set: set of pages that a process is currently using

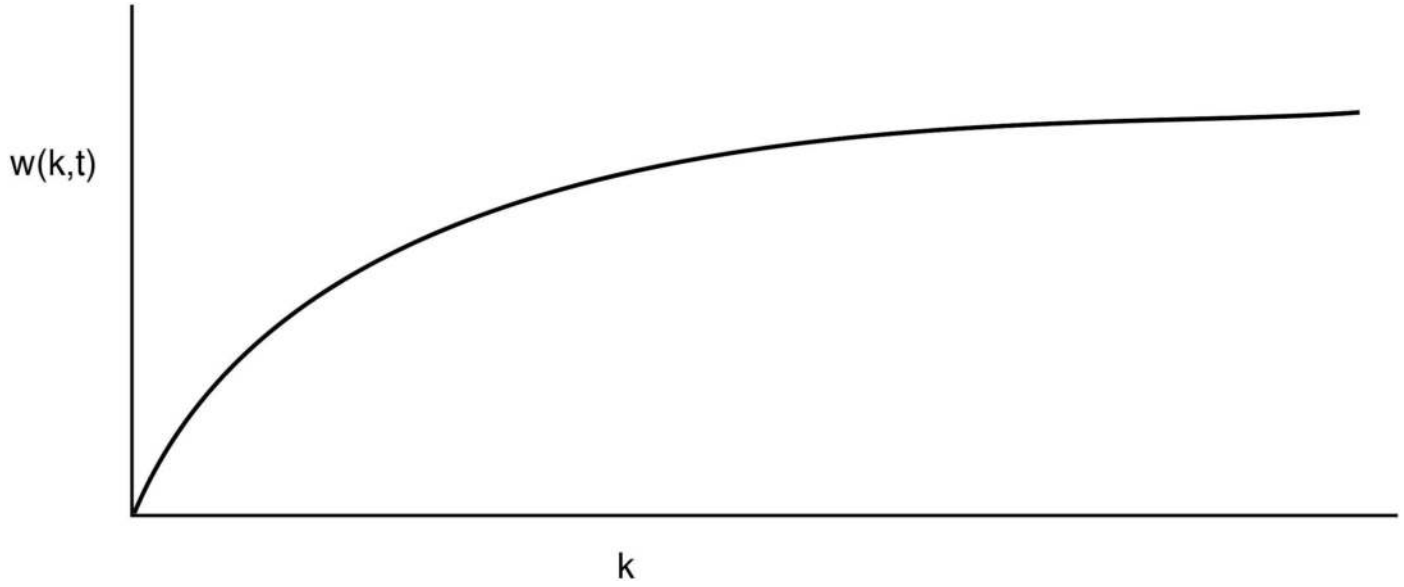


Figure 3-18. The working set is the set of pages used by the **k** most recent memory references. The function **w(k, t)** is the size of the working set at time **t**.

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Working Set

If (WS in memory) → few page faults
Else → many page faults

If (memory too small for WS) → **thrashing**
Most programs exhibit **locality of reference**

Maybe ensure pages in WS not replaced?
Maybe ensure WS is in memory by prepaging?

Pearson
Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Working Set Algorithm

WS = all pages referenced last τ seconds of virtual time (time consumed by process)

τ spans multiple clock ticks

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Working Set Algorithm (2 of 2)

NOTE: at periodic clock interrupt we clear R

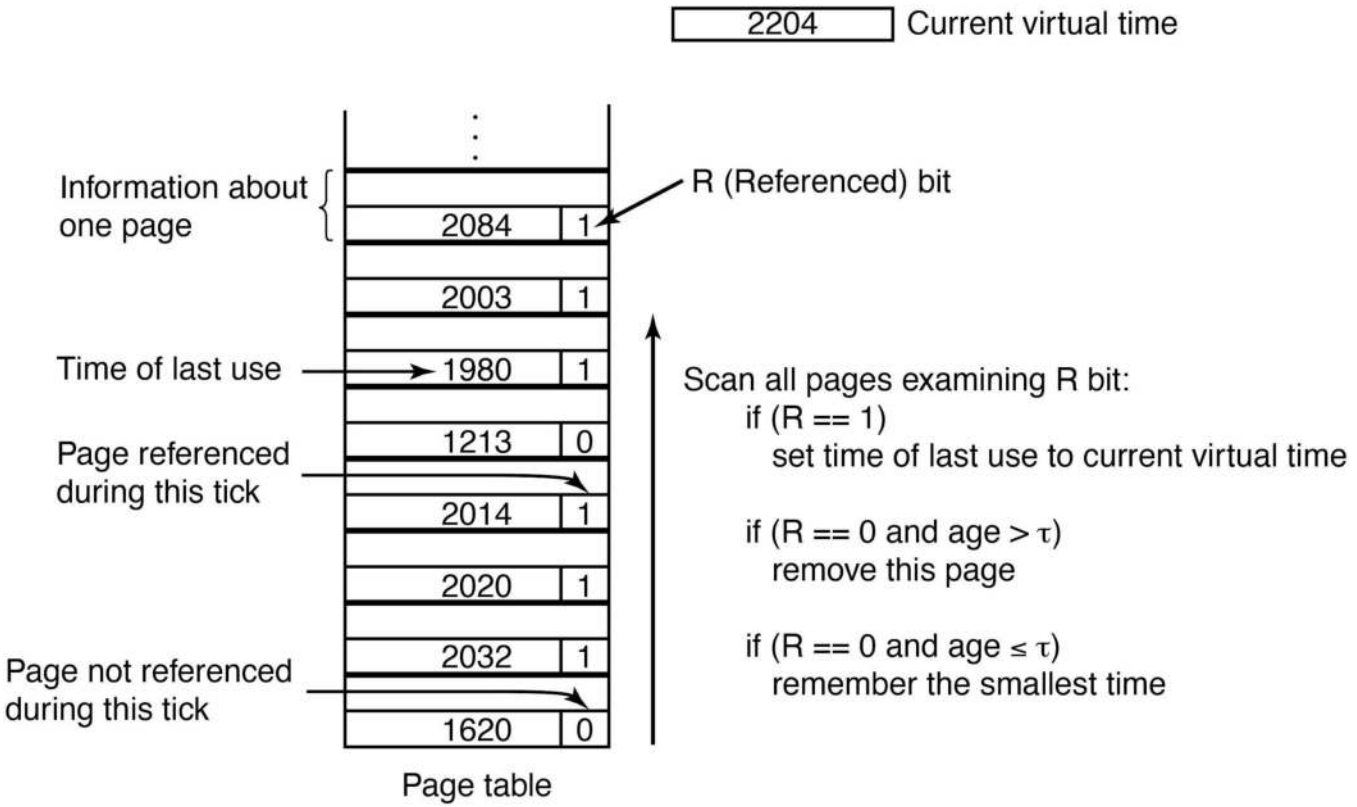


Figure 3-19. The working set algorithm.

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

WSClock

- Previous model is cumbersome as we must scan the page tables for each PF
- WSClock combines Clock and WS and is more efficient

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

WSClock

- Previous model is cumbersome as we must scan the page tables for each PF
- WSClock combines Clock and WS and is more efficient
  - Circular list of page frames (initially empty)
  - When page referenced→ add to list with time
  - Advance hand + check if R==0 (else R→0, skip after updating time)
  - If (R==0 && age > t && not dirty) → simply replace
  - If (R==0 && age > t && dirty) schedule write and cont.
  - If hand comes all the way around → 2 cases:
    - At least one page scheduled for write
    - No page scheduled for write → any clean page (if available)
      → otherwise current page``

Pearson
Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

WSClock Algorithm (1 of 2)

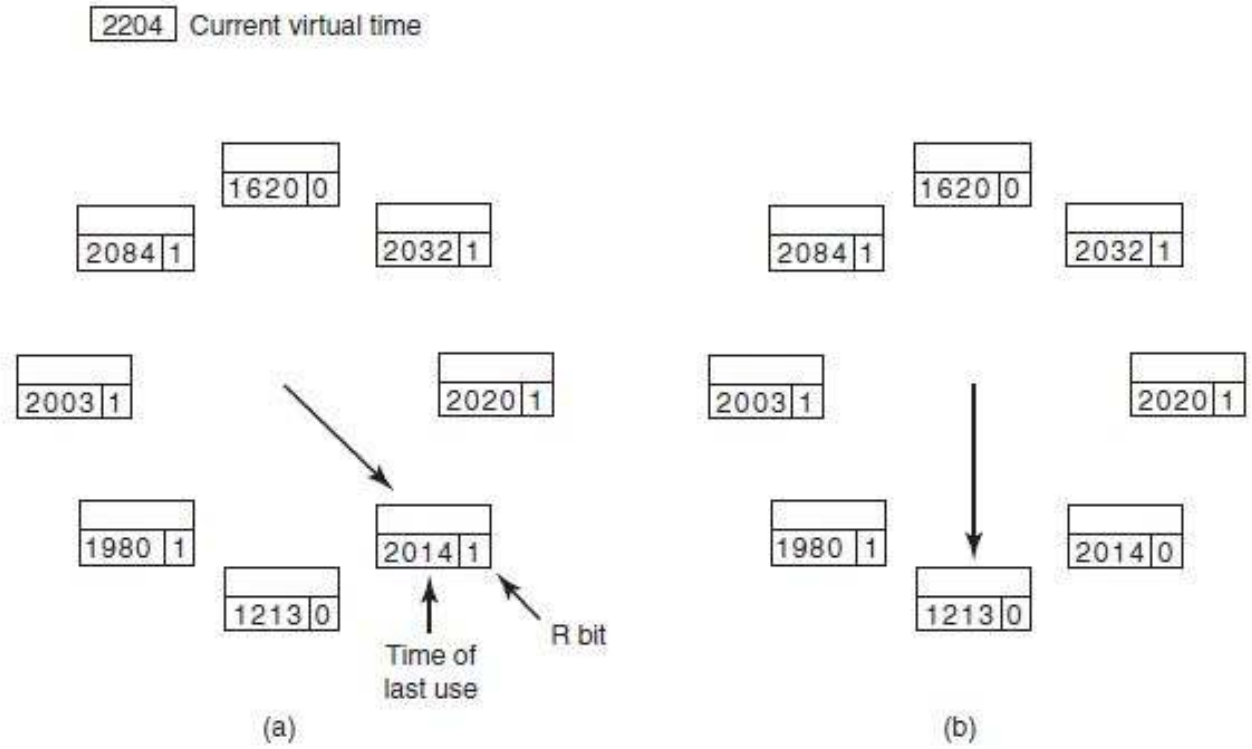


Figure 3-20. Operation of the WSClock algorithm. (a) and (b) give an example of what happens when R = 1.

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

WSClock Algorithm (2 of 2)

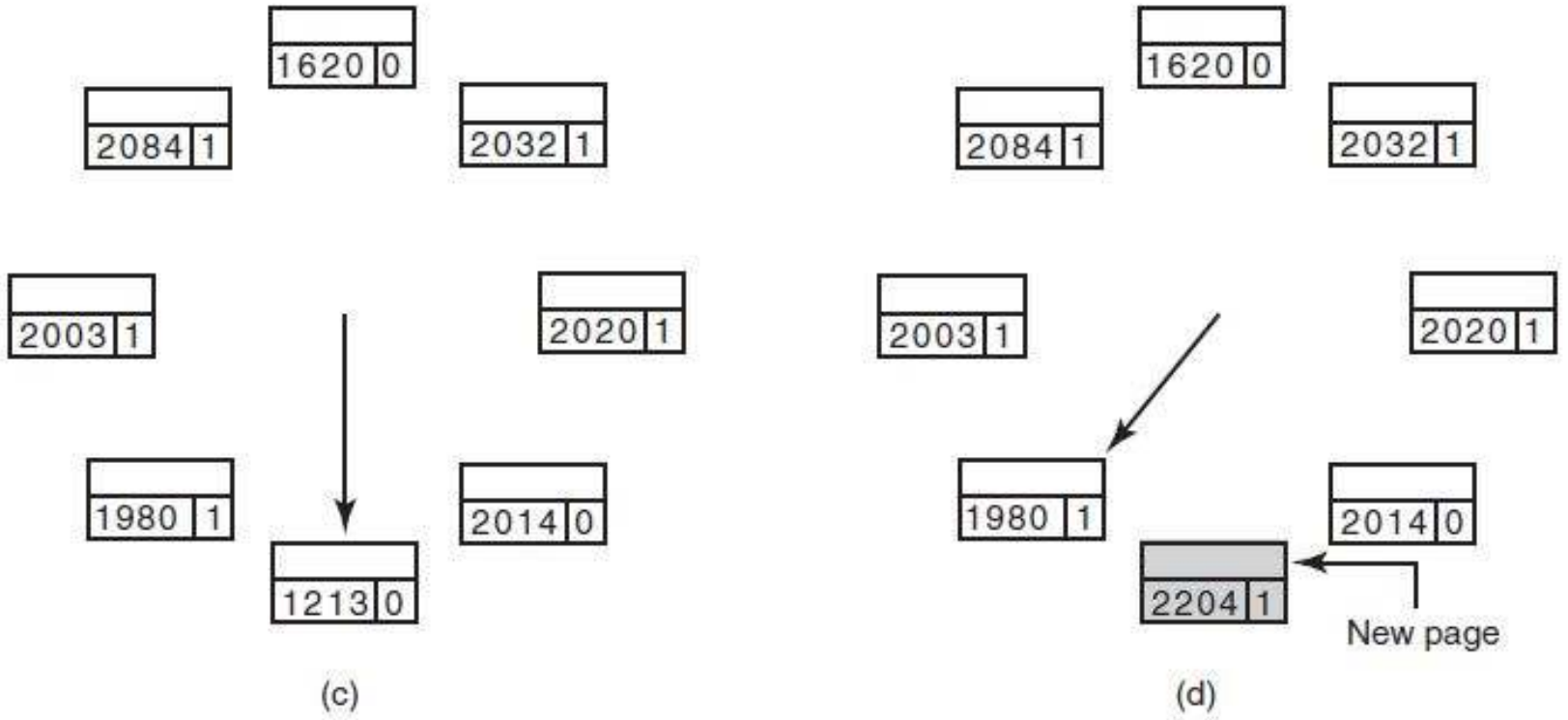


Figure 3-20. Operation of the WSClock algorithm. (c) and (d) give an example of R = 0.

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

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.

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Design Issues: Paging Strategy

- **Demand paging:**
  - Lazily allocate pages to each process
  - Exploits temporal locality to avoid frequent PFs

- **Prepaging**:
  - Typically based on the **working set** model, i.e., the set of pages a process currently needs to execute
  - Load the process working set in memory in advance when scheduling the process to avoid frequent PFs
  - Need for **working set estimation** algorithms

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Allocating memory

Extra material

Typical memory address space (Linux x86-64)

What happens if we allocate memory?

Different library functions:

`malloc` (+ friends) vs `mmap`

Different syscalls:

`brk, sbrk` vs `mmap`

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

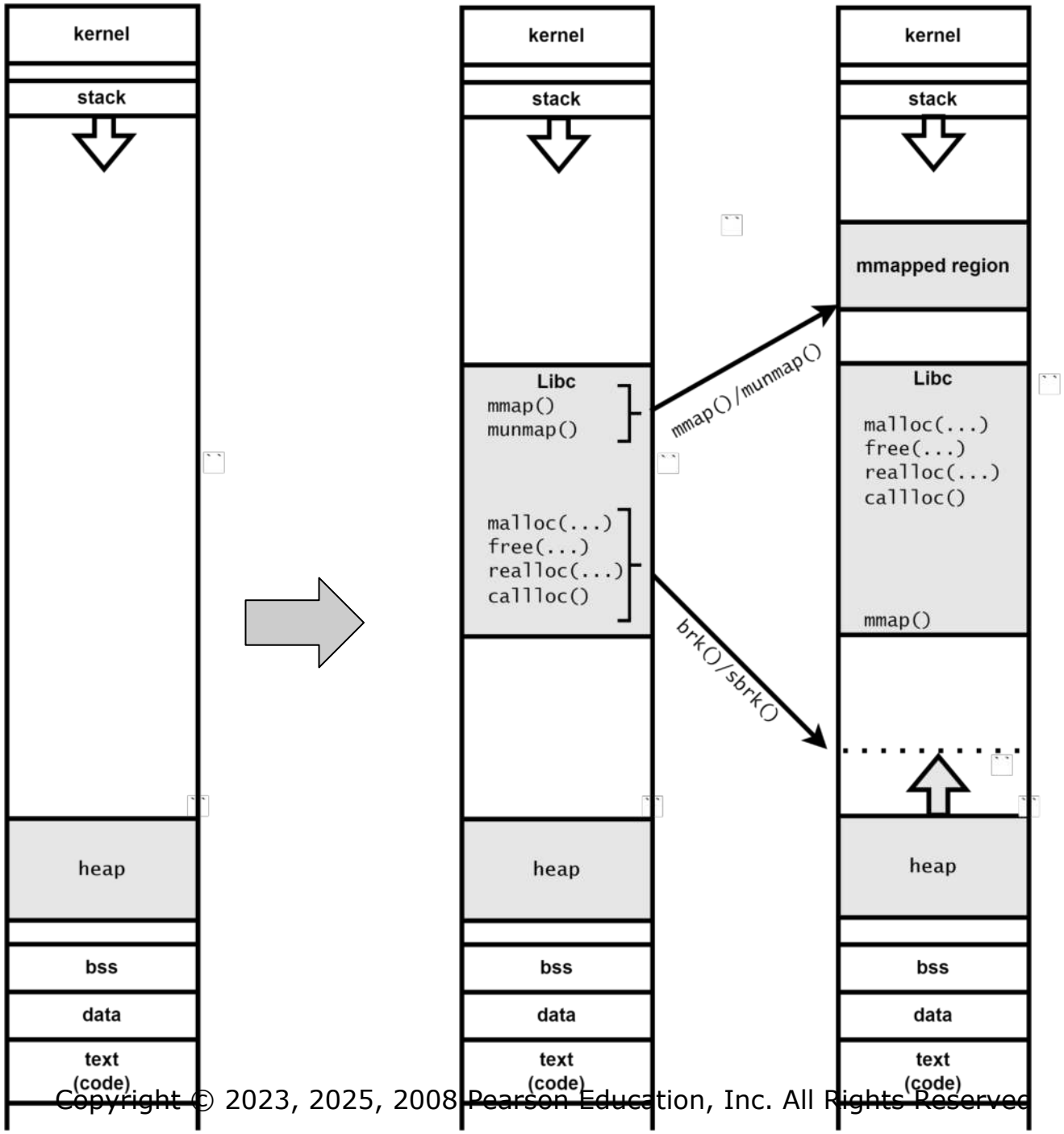
Allocating memory

Extra material

On Linux, `malloc()` results in

- `(s)brk()` syscall for small areas
- `mmap()` syscall for large areas

`mmap()` always results in `mmap()`



Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Allocating memory

Extra material

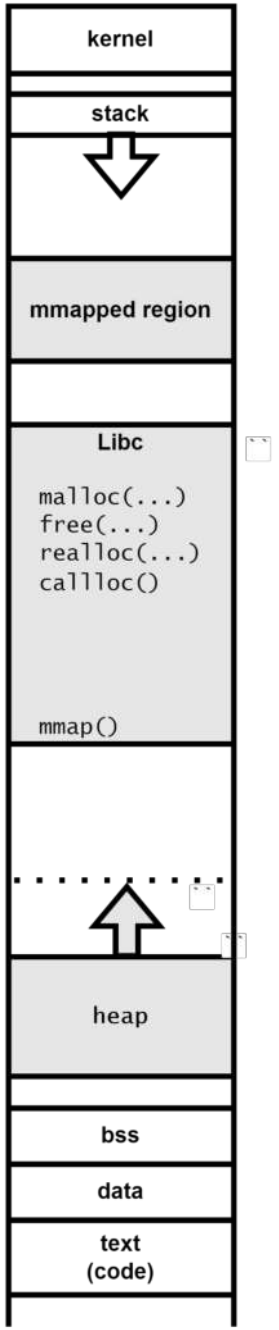
For all of the memory holds:

- access a location in memory ⇒ MMU
- may trigger PF
- may trigger SEGV

For instance, Linux maintains a doubly linked list of VMAs (Virtual Memory Areas) indicating which regions are allocated

Allocating virtual memory vs assigning physical memory

- OS may “allocate memory” (enter it as *used* in its administration) without assigning physical memory to it yet
- A physical page frame is assigned only when the program *accesses* a location on the virtual page → **demand paging**



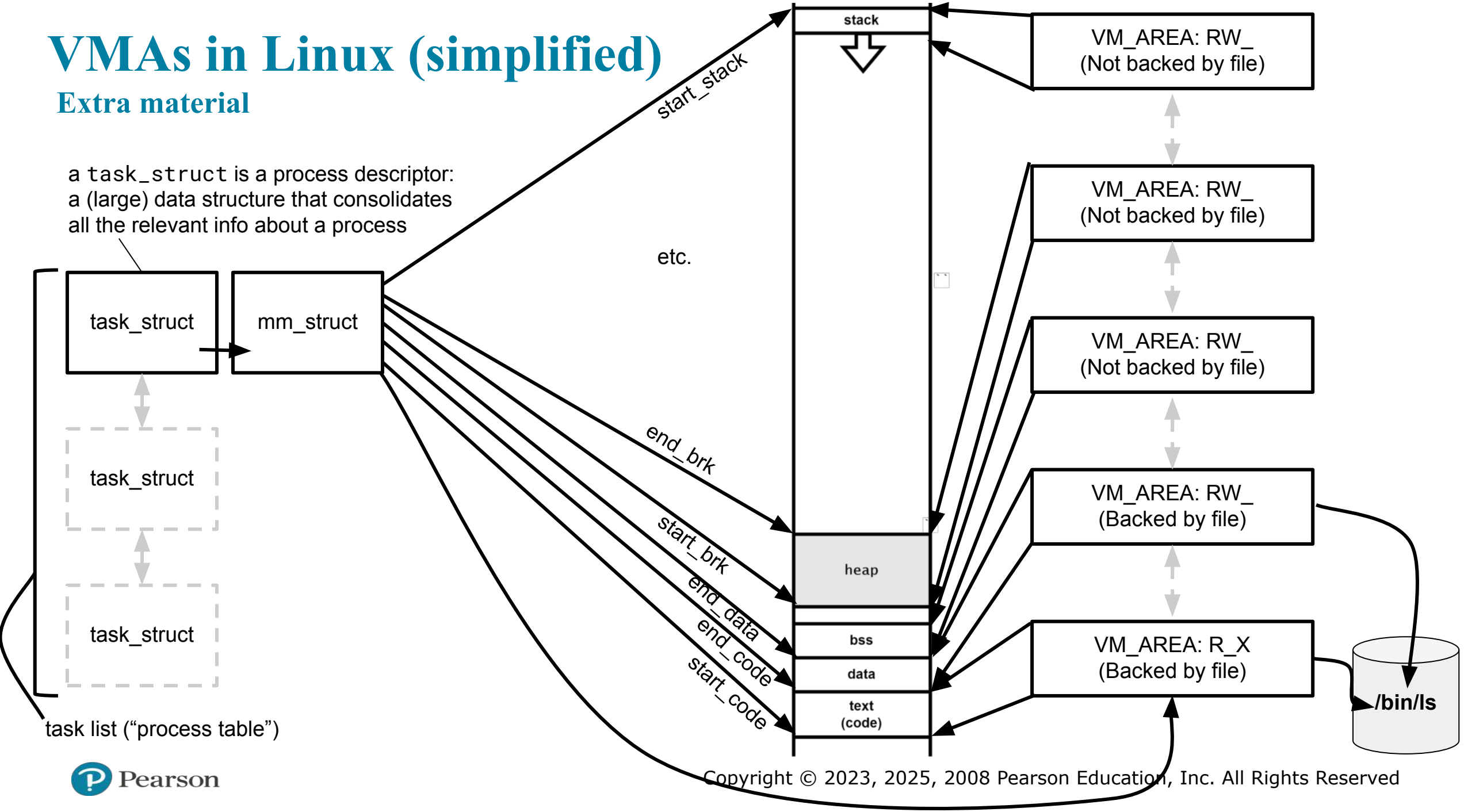
Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

VMAs in Linux (simplified)

Extra material

a `task_struct` is a process descriptor:
a (large) data structure that consolidates
all the relevant info about a process

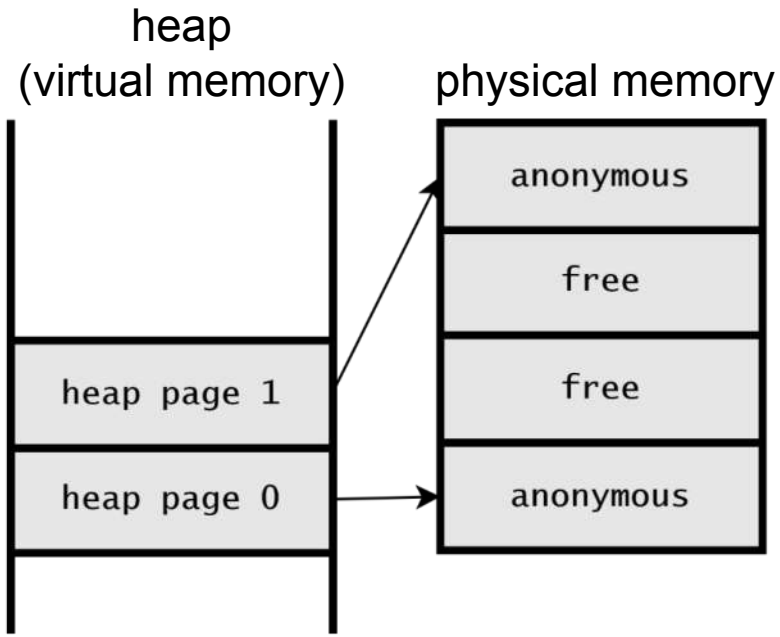


Pearson

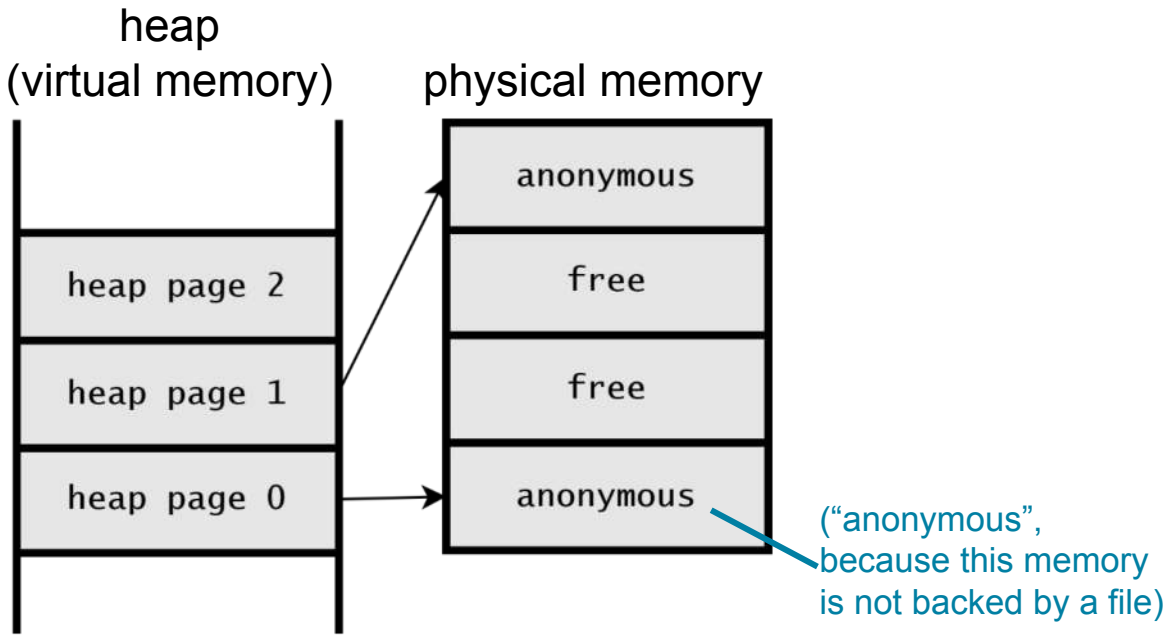
Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Demand paging (1/2)

Extra material



The program extends the heap with brk()



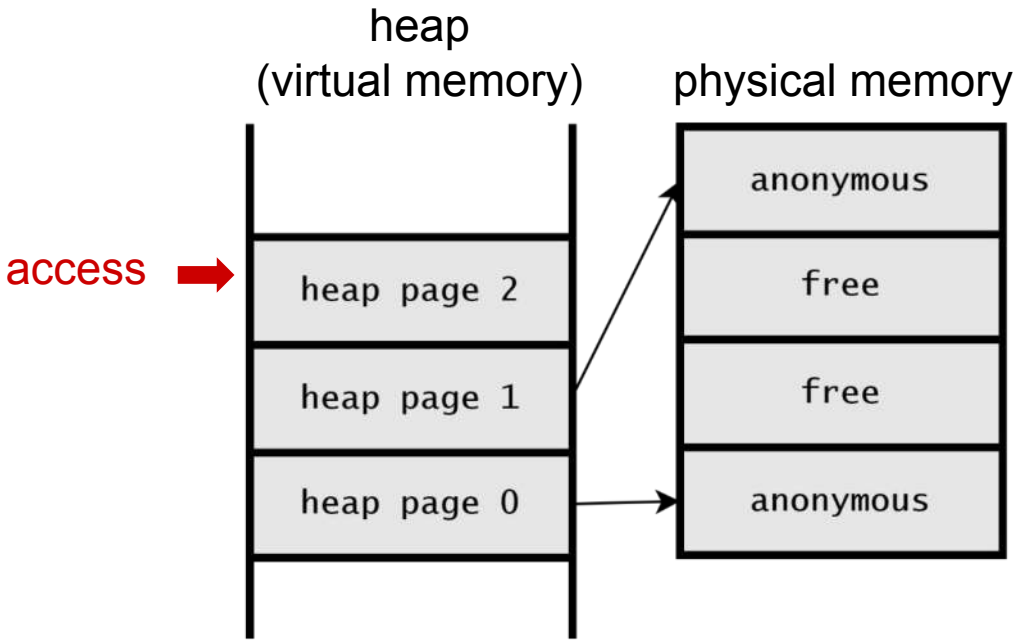
The heap is now extended, but no physical frame is assigned to page 2

Pearson

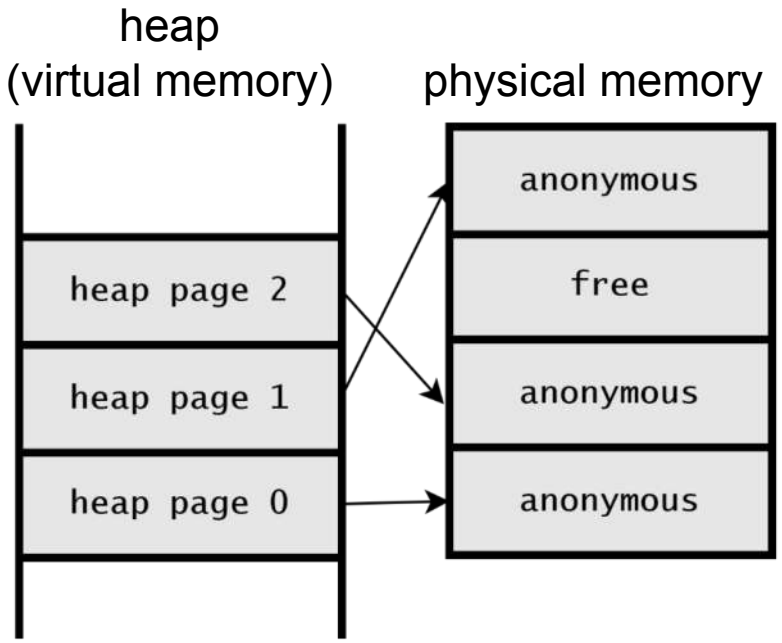
Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Demand paging (2/2)

Extra material



Now the program reads from or writes to the new page on the heap → PF



The OS maps a new physical frame to back the virtual page.

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Demand Paging: Linux Allocation Benchmark

Extra material

```c
#include <assert.h>
#include <stdlib.h>

int main(int argc, char* argv[])
{
    void *ptr;
    int i, size;
    assert(argc == 2);
    size = atoi(argv[1]);

    for (i=0;i<1000000;i++) {
        ptr = malloc(size);
        assert(ptr);
        free(ptr);
    }

    return 0;
}
```

→ Note: we allocate and free immediately → no accesses (demand paging, so not backed by physical memory)

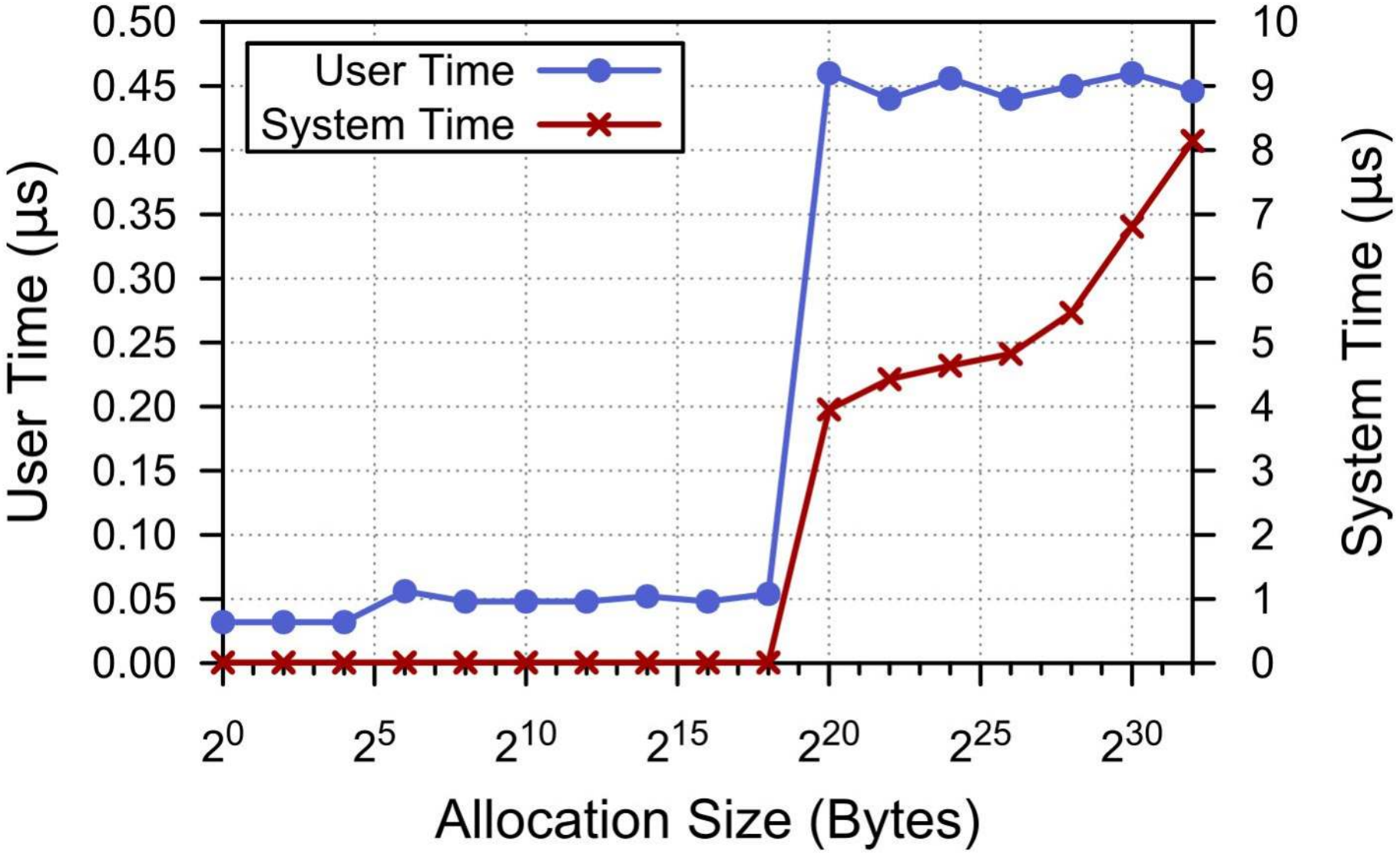
We run this code and plot the time

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Demand Paging: Linux Allocation Benchmark

Extra material



Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Design Issues: Allocation Policies

- **Local allocation:**
  - Replaces only pages of the current process
  - Assumes static process memory allocation size
  - Problems:
    - A growing **working set** leads to **trashing**
    - A shrinking working set leads to wasted memory
- **Global allocation**:
  - Replaces pages owned by any process. Strategies:
    - Treat every global page equally (or prioritize)
    - Dynamic memory balancing using **working set size** estimation
    - Selectively swap out processes (or OOM kill)

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Design Issues:
Local versus Global Allocation Policies (1 of 2)

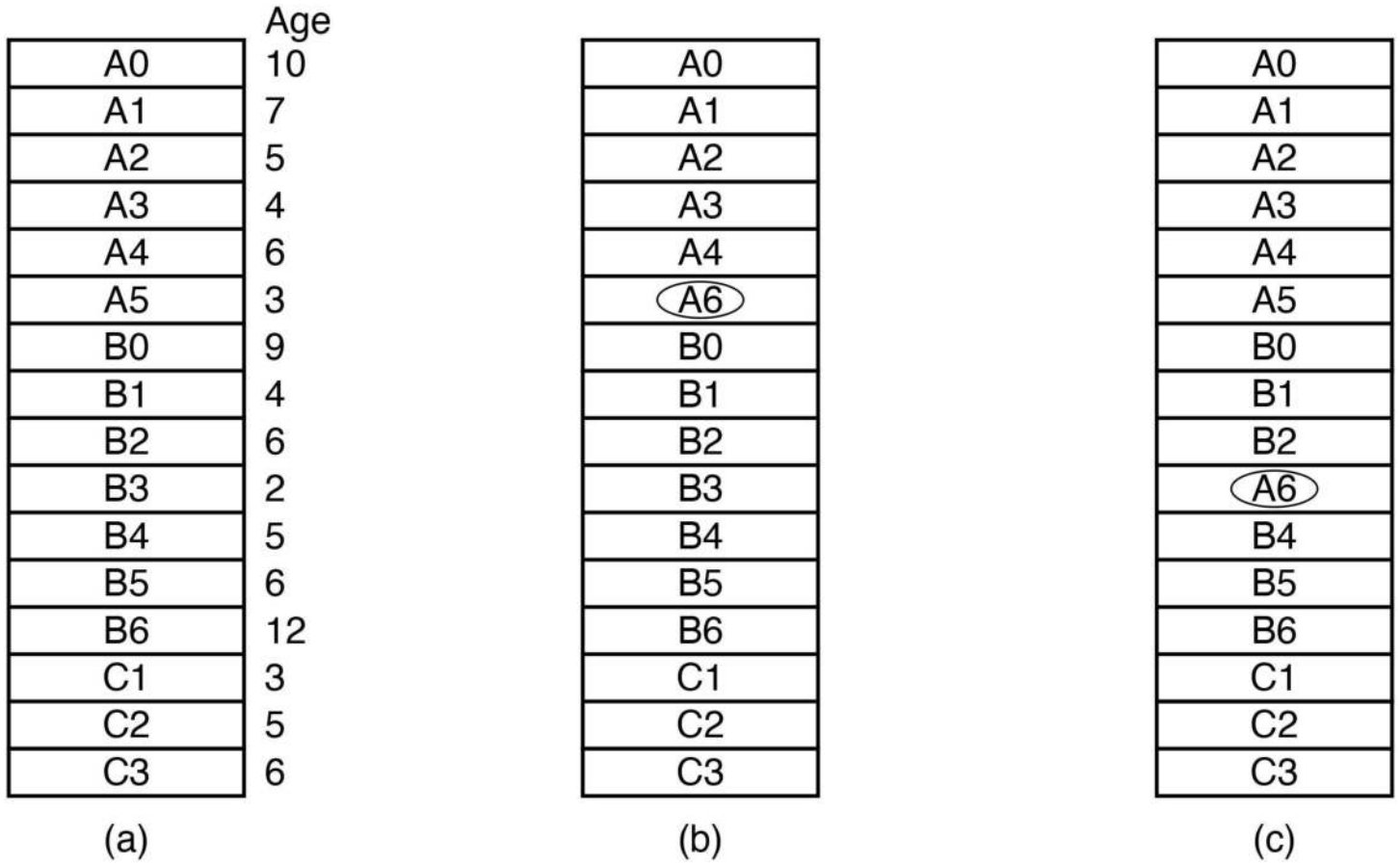


Figure 3-22. Local versus global page replacement. (a) Original configuration. (b) Local page replacement. (c) Global page replacement.

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Design Issues:
Local versus Global Allocation Policies (2 of 2)

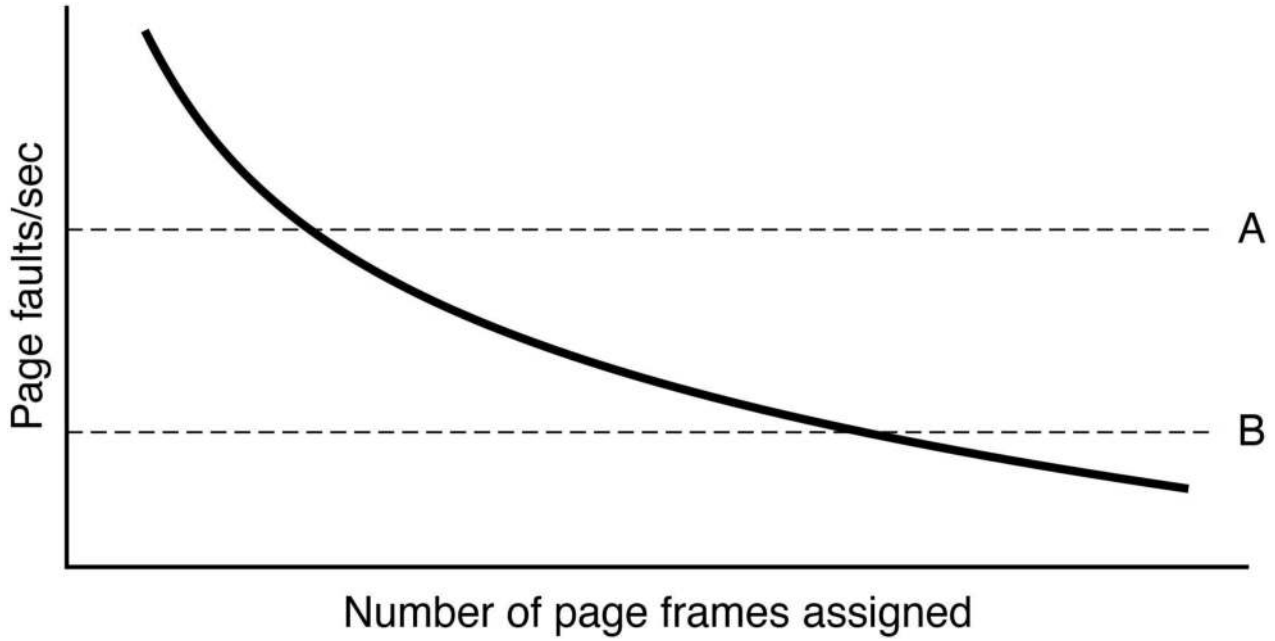


Figure 3-23. Page fault rate as a function of the number of page frames assigned.

Pearson
Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Design Issues: Load Control

- Thrashing can be expected when the combined working sets of all processes exceed the capacity of memory
- Out of Memory killer (OOM) selectively kills processes when system is low memory
- Less drastic approach swaps some processes to nonvolatile storage and releases those pages
  - Similar to two-level scheduling
- Can also reduce memory usage via compaction/compression
  - Deduplication or same page merging

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Design Issues: Page Size

Most paging systems support 4KB pages

relatively small : limits internal fragmentation / space wastage

not too small : limits number of pages to handle, entries in TLB

In addition, many support also larger pages. For instance:

2 MB

1GB

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Design Issues:
Separate Instruction and Data Spaces

Most computers have a single address space shared by program and data
Ìn the past some systems had a separate address space for instructions and data

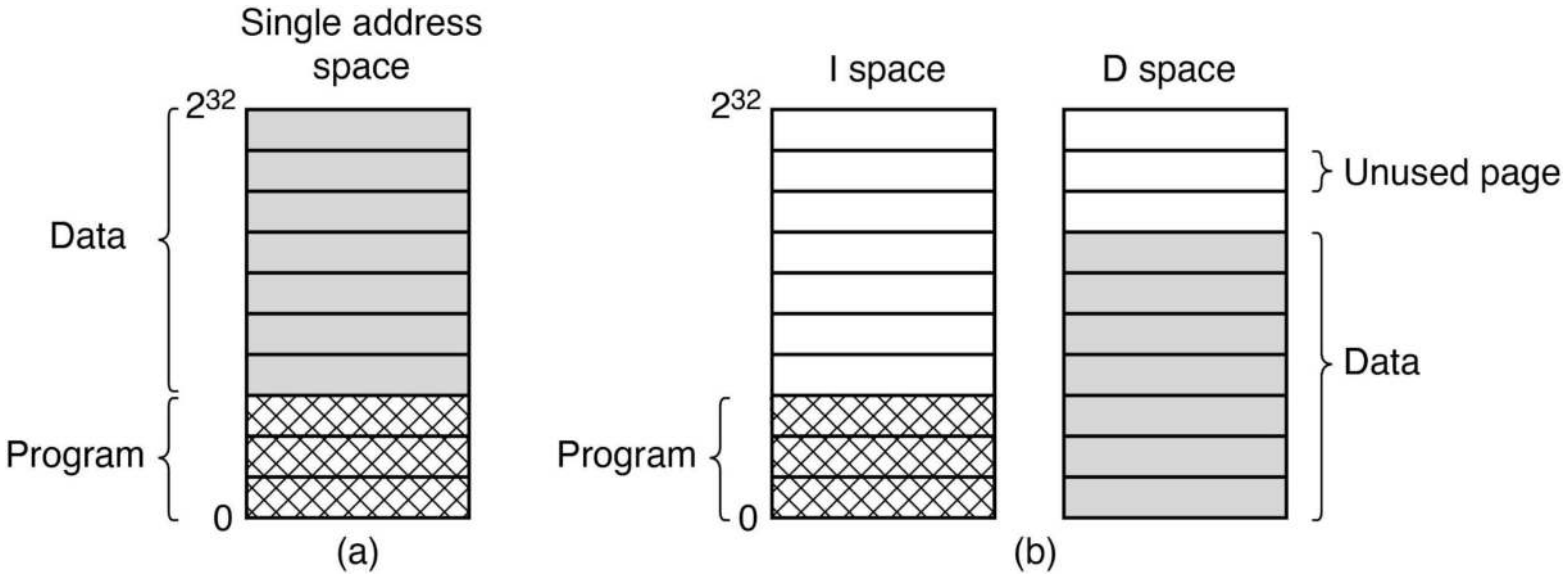


Figure 3-24. (a) One address space. (b) Separate I and D spaces.

Nowadays, we still see separate I and D spaces in caches and TLBs, etc.

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Design Issues: Shared Pages

- Sharing improves efficiency.
- When one process exits, OS should realise there is another user of the the page

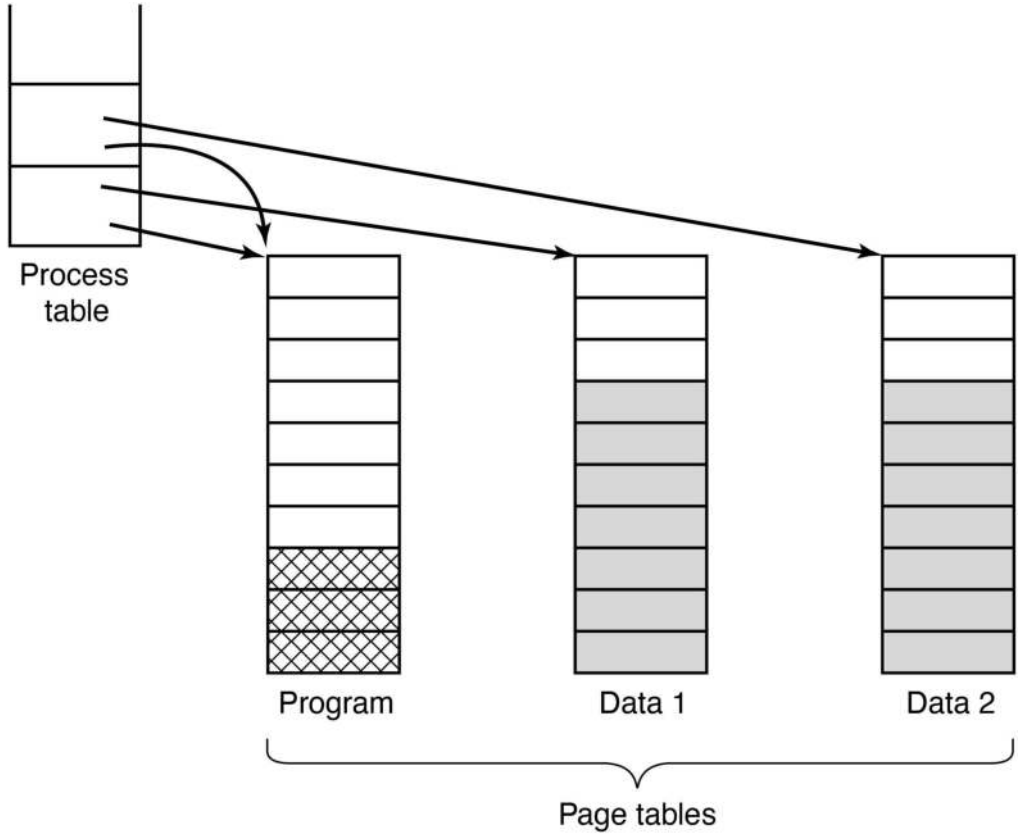


Figure 3-25. two processes sharing same program sharing its page table.

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Design Issues: Shared Pages

- Sharing improves efficiency.
- When one process exits, OS should realise there is another user of the the page

Sharing data is trickier than sharing code

- After `fork` system call parent and child share program and text
  - To do so efficiently, each gets its own set page table, pointing to the same pages
  - Pages are mapped read-only
  - When a process writes to such a page: make a copy
  - This is known as **copy-on-write**

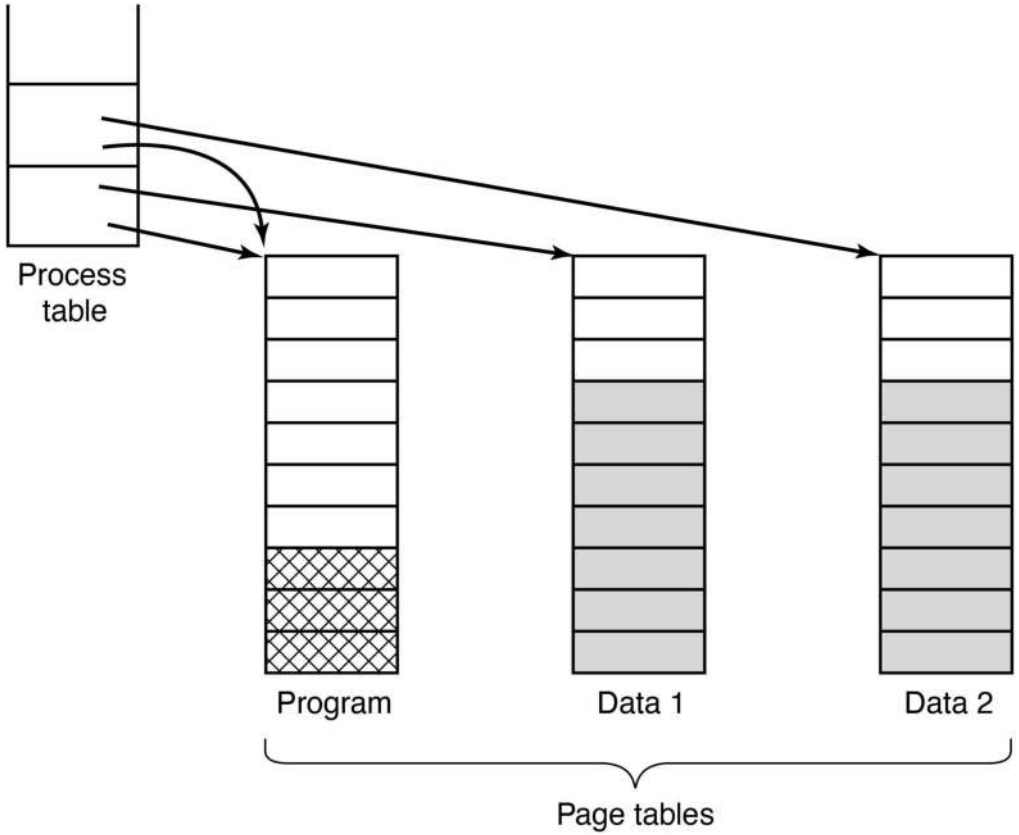
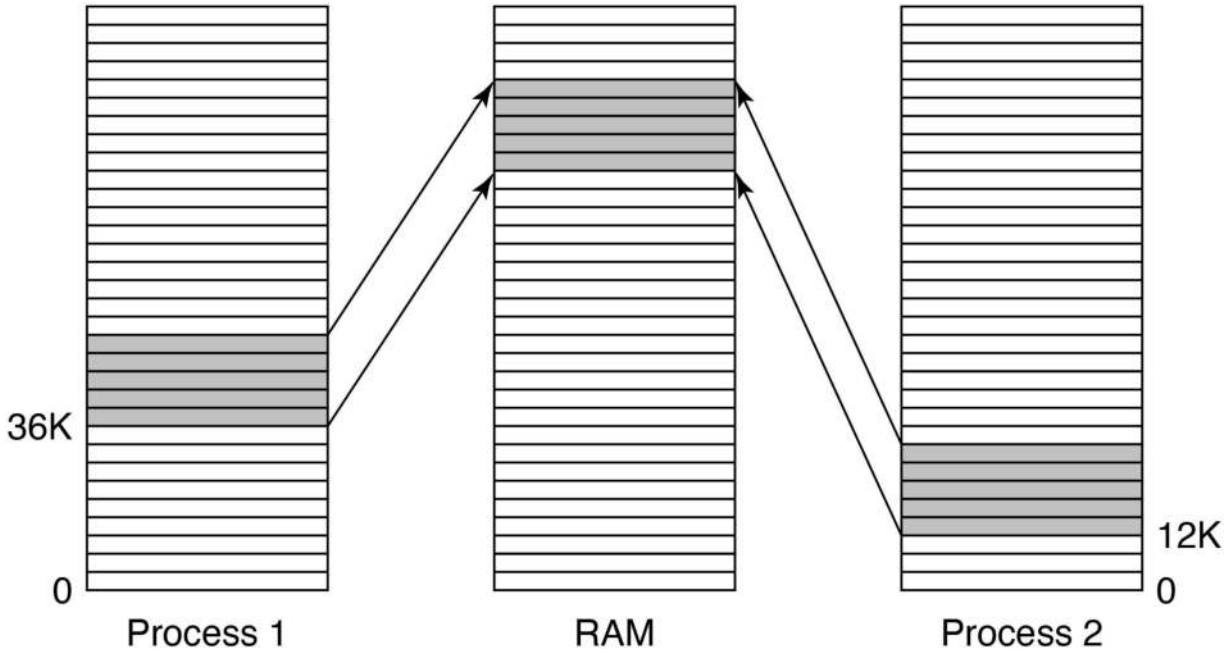


Figure 3-25. two processes sharing same program sharing its page table.

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Design Issues: Shared Libraries



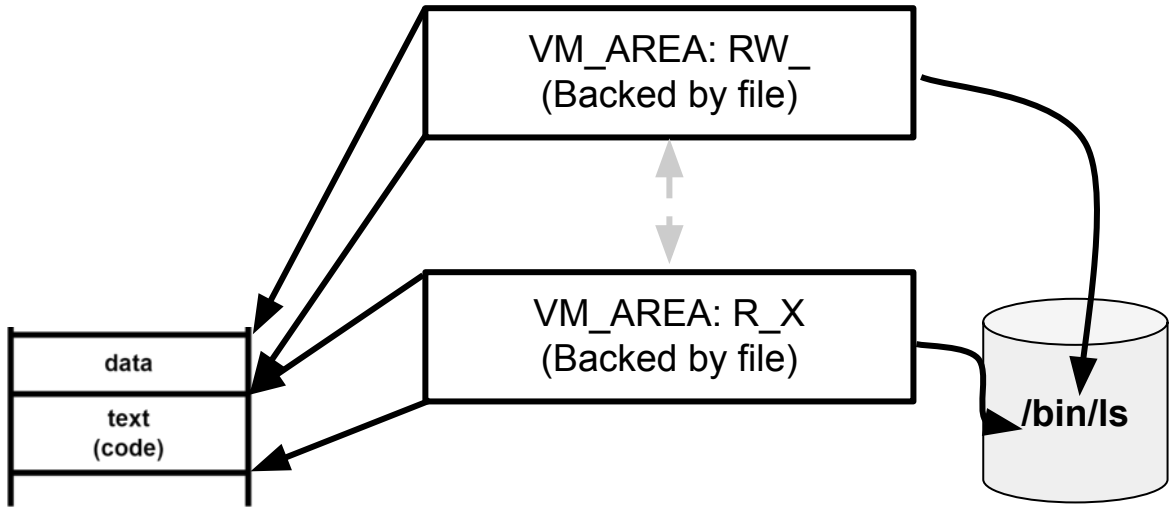
- Sharing libraries is very common
  - easy: read-only
- Typically through Dynamic Link Libraries (DLLs)
  - often as position-independent code

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

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Design Issues:
Memory mapped files



Mapped file

Shared libs special cases of memory mapped files

As pages are touched → demand paged from disk

When process exits → write dirty pages back to disk

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Design Issues: Cleaning policy

Paging works well if plenty of free pages. What if we run short?

Indirect reclaim: Paging daemon sleeps and then evicts pages if free pages are in short supply

Leaves them in memory -- easy reuse

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Design Issues: Virtual Memory Interface

- Allocator interface
  - `malloc` family, `mmap` family
- Memory-mapped files
  - Treat files as memory objects for faster I/O
  - Facilitate code sharing for programs/shared libraries
- Copy-on-write semantics
  - Lazy copying for `fork()`, deduplication, checkpointing, etc.
- Shared memory
  - MAP_SHARED mappings, key-based, file-based
- Distributed shared memory
  - Shared memory semantics over the network

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Page Fault Handling (1 of 3)

1. The hardware traps to kernel, saving program counter on stack.
2. Assembly code routine started to save registers and other volatile info. Calls the page fault handler
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

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Page Fault Handling (2 of 3)

5. If frame selected dirty, page is scheduled for transfer to nonvolatile storage, 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 or SSD operation to bring it in.
7. When disk or SSD interrupt indicates page has arrived, tables updated to reflect position, and frame marked as being in normal state.

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Page Fault Handling (3 of 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 where it left off

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Separation of Policy and Mechanism (1 of 2)

Again: the separation is important for managing the complexity of the system.

Not always trivial. Example:

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.

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Separation of Policy and Mechanism (2 of 2)

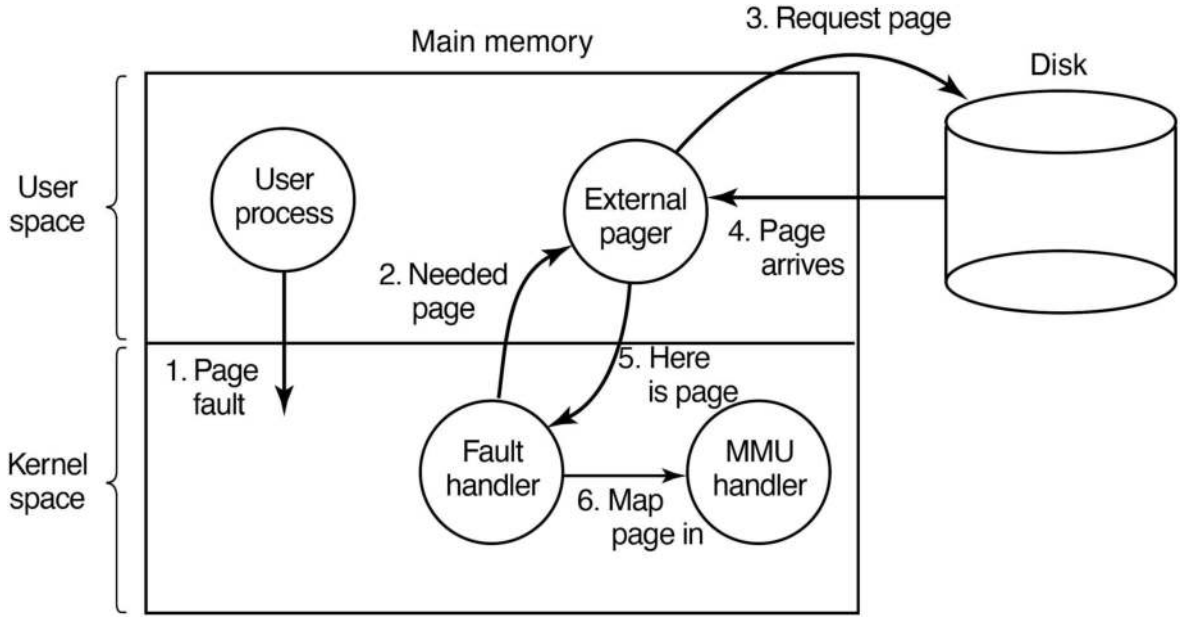


Figure 3-29. Page fault handling with an external pager.

- Kernel page fault handler:
  - machine independent
  - contains *mechanism* for paging
- External pager:
  - machine independent
  - can implement the *policy*
- Not always easy. For instance: cleanest if replacement algorithm in external pager
  - But may not have access to R and M bits
    - either need new mechanism to pass these bits to user space
    - or replacement algorithm in kernel

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Segmentation: Paging is not the only option

Motivation: many programs have many dynamic areas.

Examples of tables generated by compiler:

1. Source text being saved for the printed listing
2. Symbol table: names and attributes of variables.
3. Table containing integer and floating-point constants used.
4. Parse tree, syntactic analysis of the program.
5. Stack used for procedure calls within compiler.

Areas may bump into each other

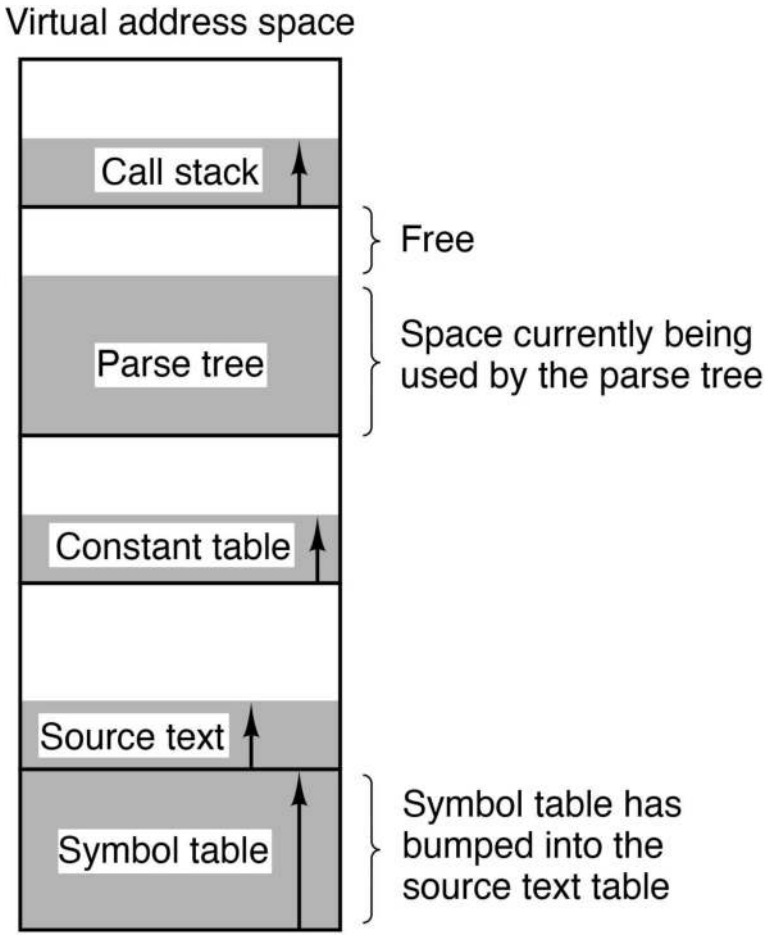


Figure 3-30. In a one-dimensional address space with growing tables, one table may bump into another.

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Segmentation: The Idea

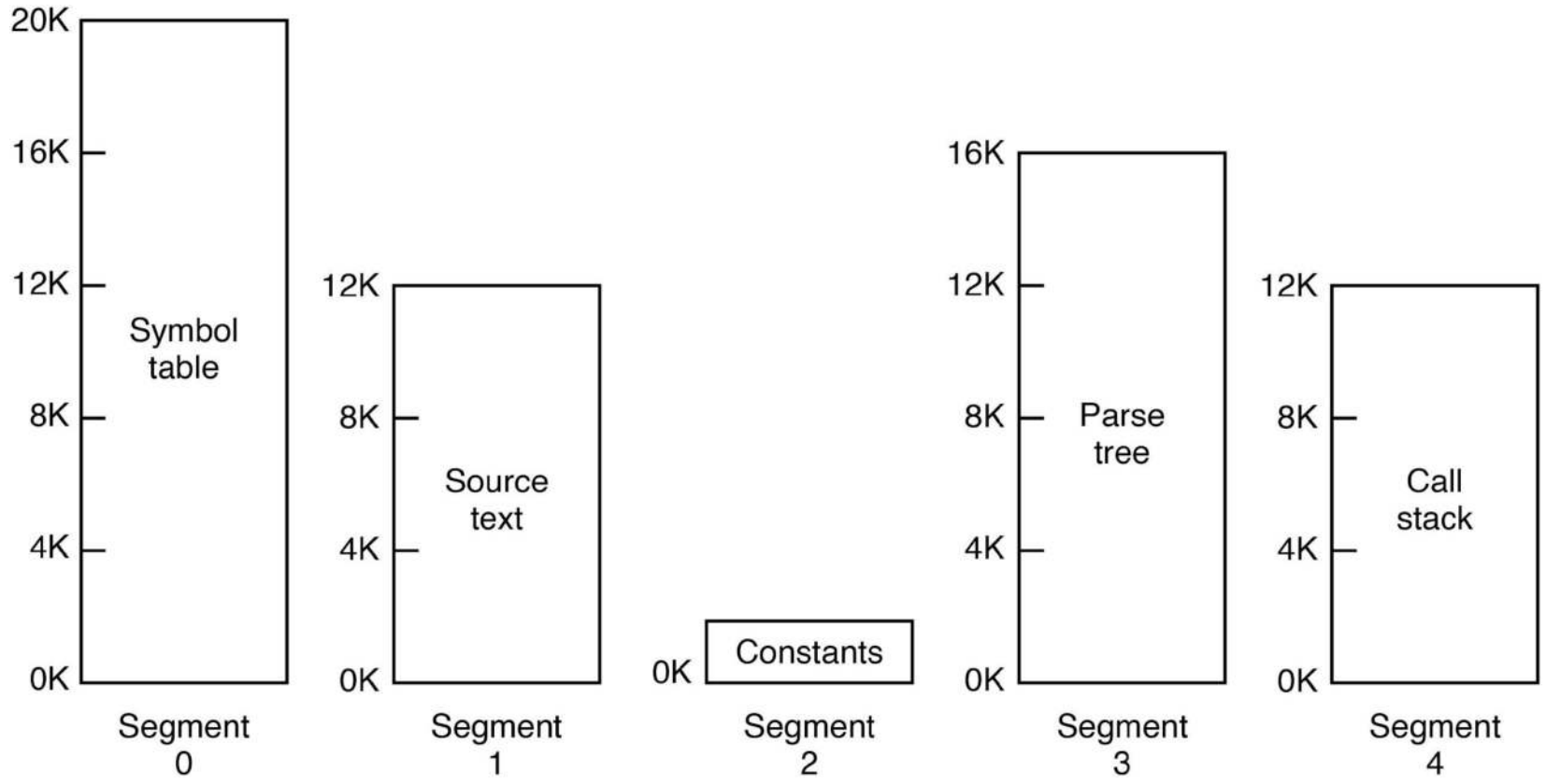
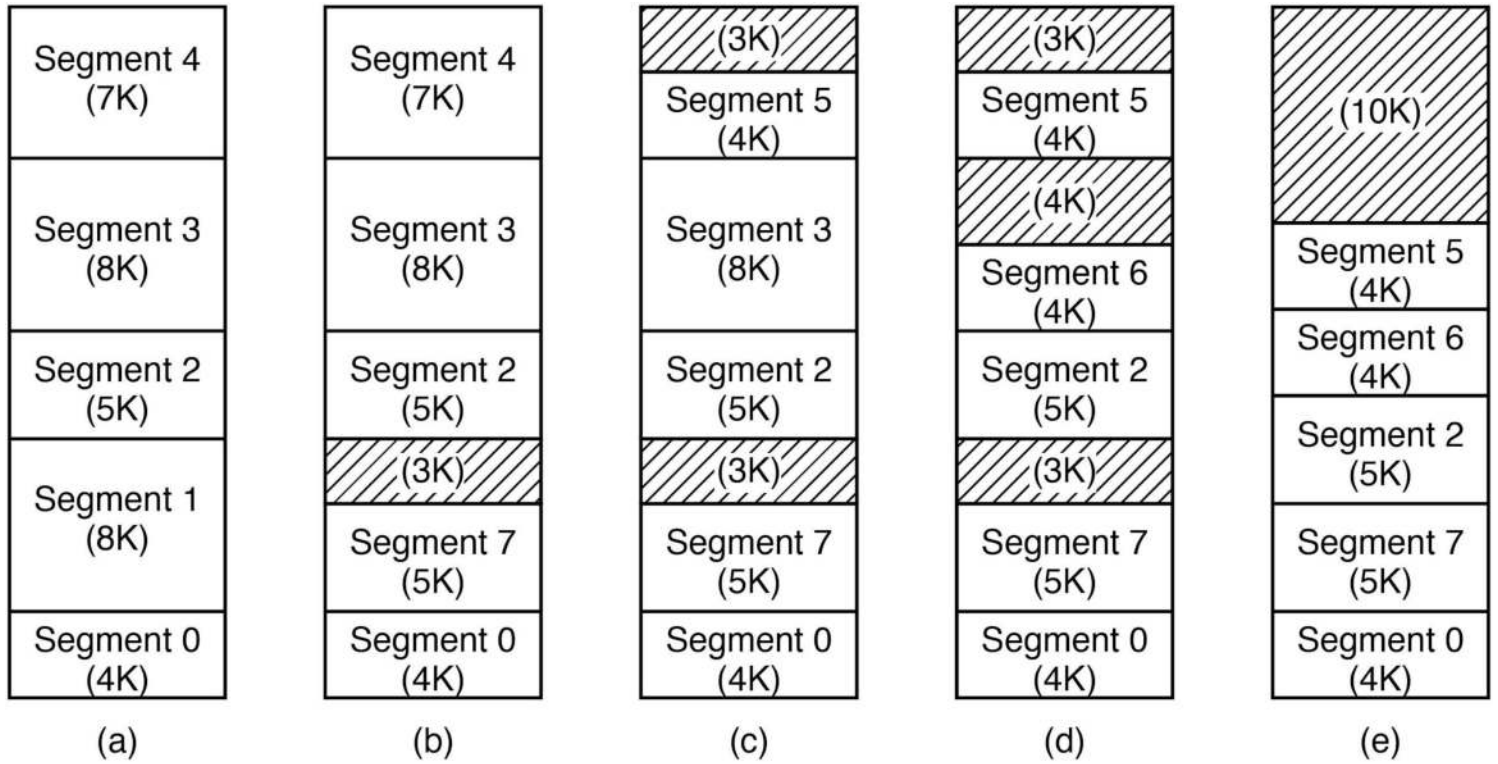


Figure 3-31. A segmented memory allows each table to grow or shrink independently of the other tables.

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Implementation of Pure Segmentation



Problem: external fragmentation

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

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Segmentation with Paging: MULTICS (1 of 5)

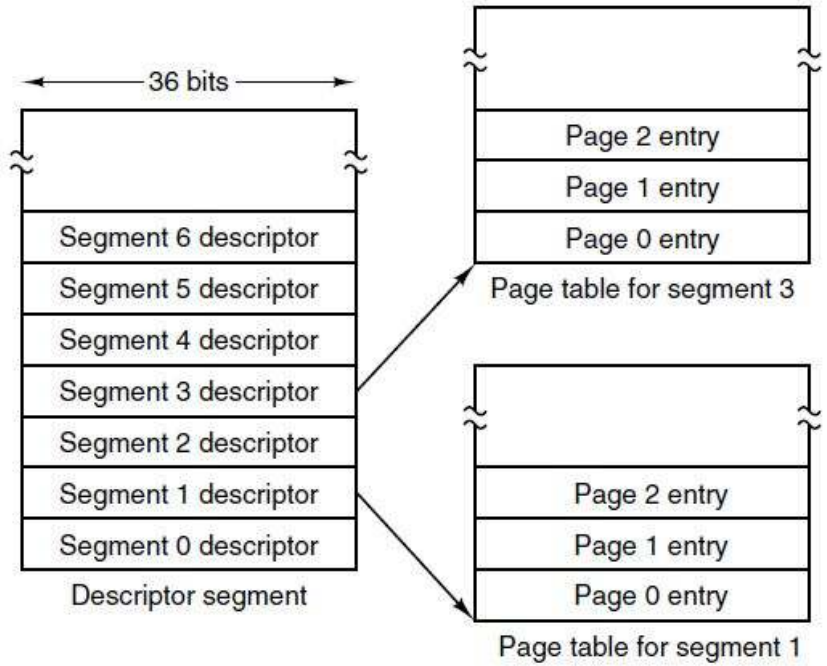


Figure 3-34. The MULTICS virtual memory. (a) The descriptor segment pointed to the page tables.

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Segmentation with Paging: MULTICS (2 of 5)

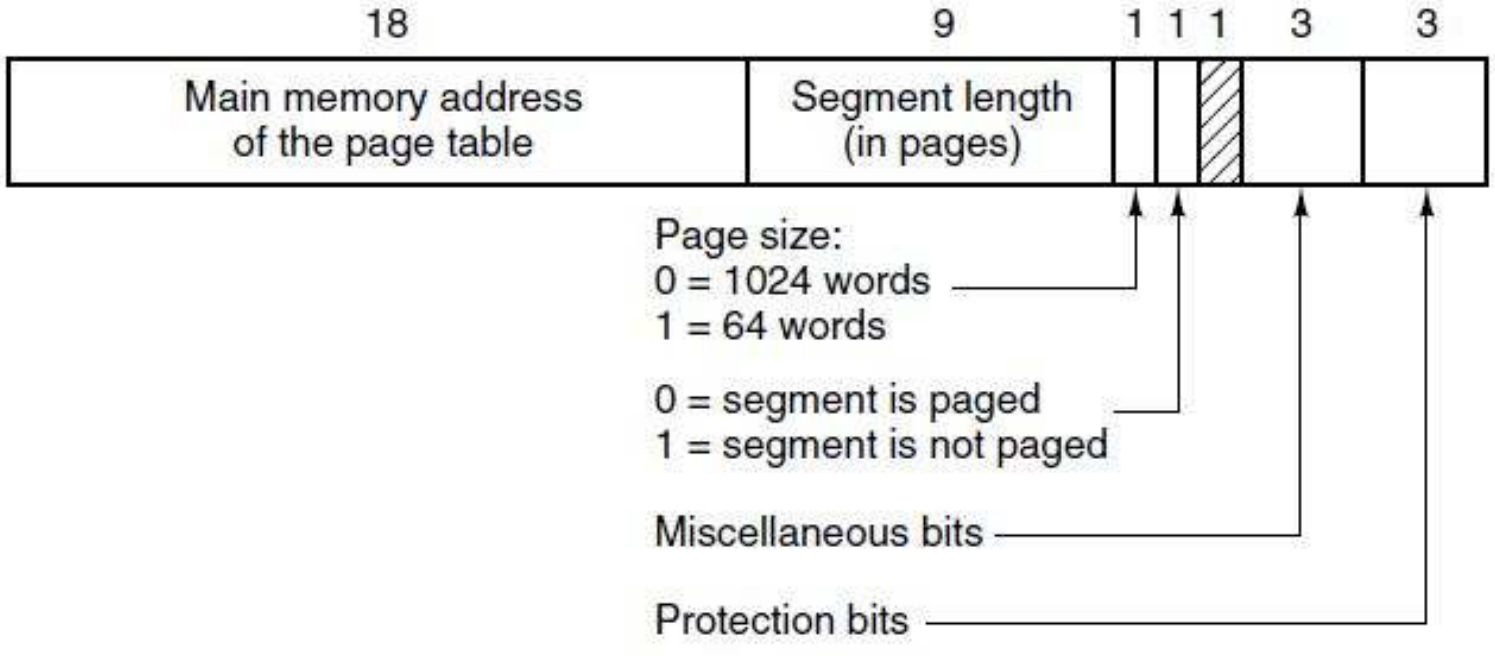


Figure 3-34. The MULTICS virtual memory. (b) A segment descriptor. The numbers are the field lengths.

Pearson
Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Segmentation with Paging: MULTICS (3 of 5)

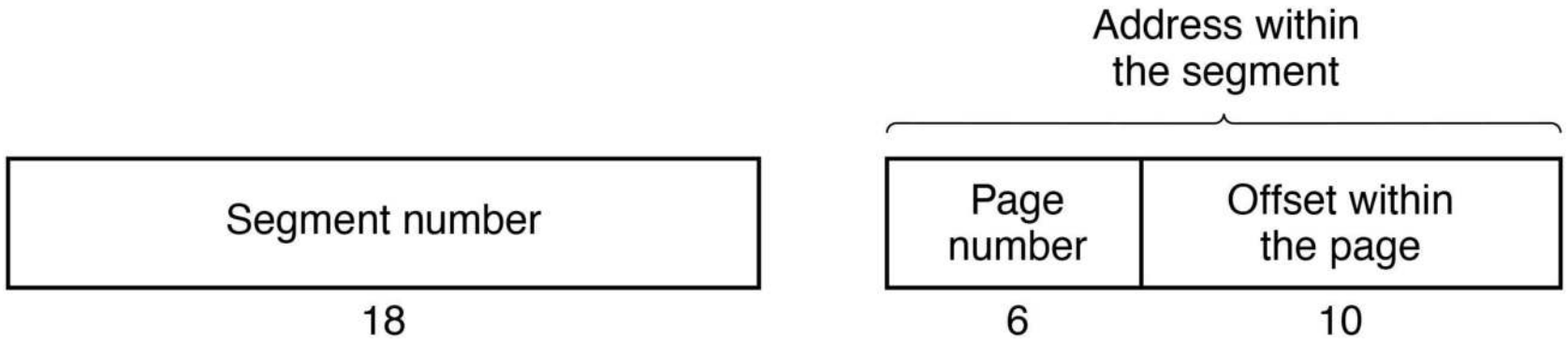


Figure 3-35. A 34-bit MULTICS virtual address.

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Segmentation with Paging: MULTICS (4 of 5)

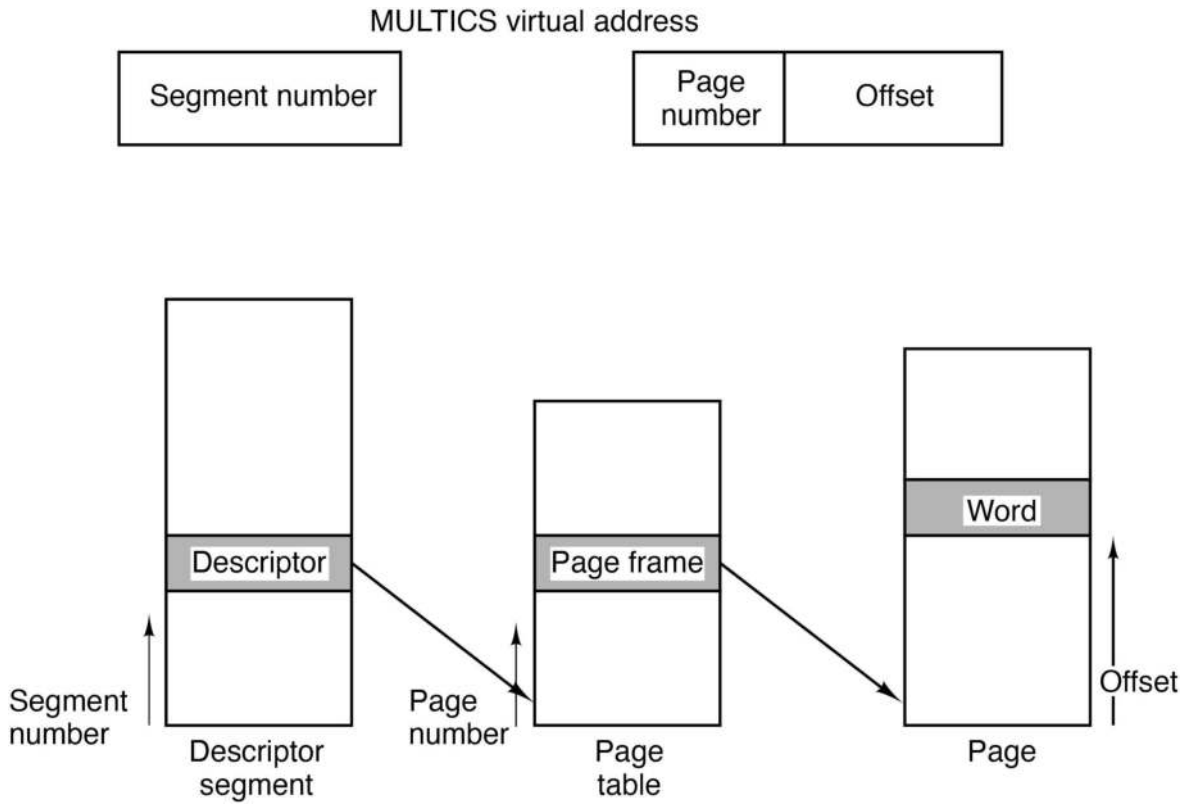
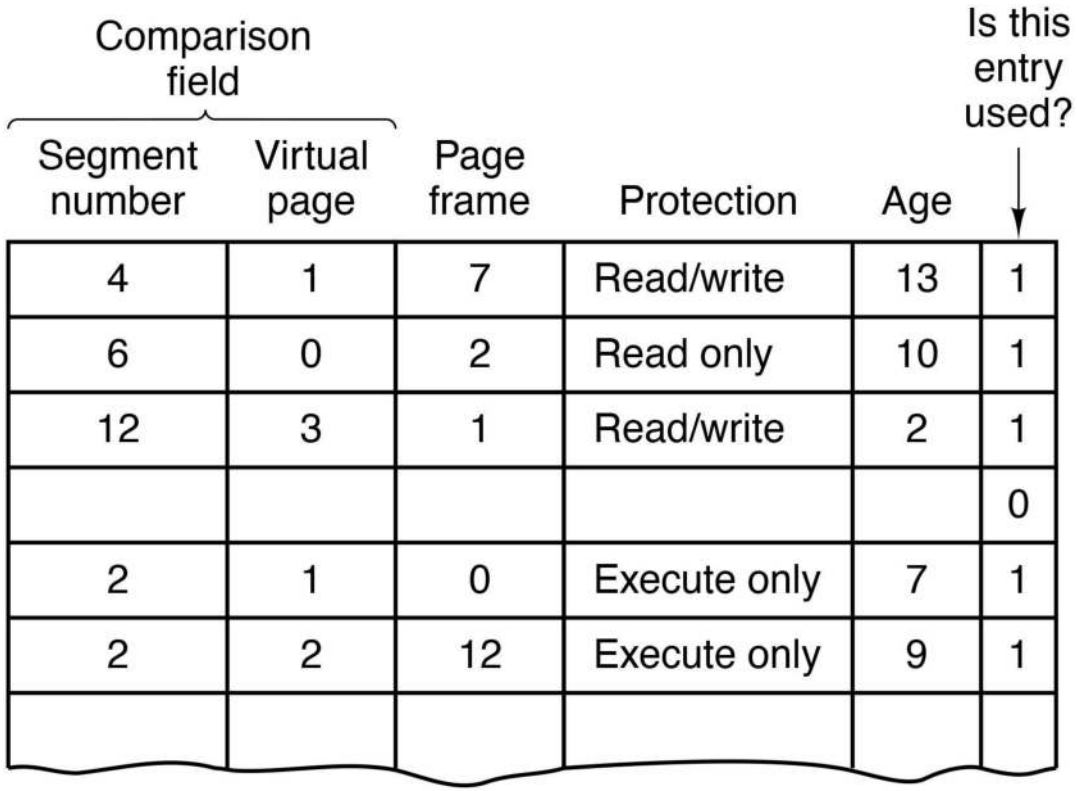


Figure 3-36. Conversion of a two-part MULTICS address into a main memory address.

Pearson
Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Segmentation with Paging: MULTICS (5 of 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.

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Segmentation (4 of 4)

Figure 3-32. Comparison of paging and 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

Pearson

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Summary

In this chapter, we looked at memory management (MM) in detail

We discussed many core concepts:

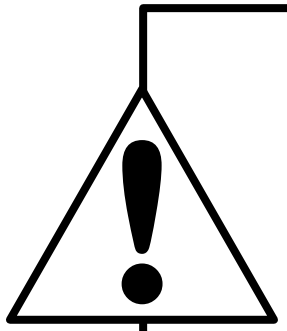
- memory abstractions
- virtual memory
- memory allocators
- paging
- page replacement
- segmentation
- design issues

On modern systems MM can be very complicated

Next, we will look at another core component: File Systems!

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved

Copyright



This Work is protected by the United States copyright laws and is provided solely for the use of instructors teaching their courses and assessing student learning. Dissemination or sale of any part of this Work (including on the World Wide Web) will destroy the integrity of the Work and is not permitted. The Work and materials from it should never be made available to students except by instructors using the accompanying text in their classes. All recipients of this work are expected to abide by these restrictions and to honor the intended pedagogical purposes and the needs of other instructors who rely on these materials.

Copyright © 2023, 2025, 2008 Pearson Education, Inc. All Rights Reserved