

# Operating Systems

Timo Hönig

Bochum Operating Systems and System Software (BOSS)

Ruhr University Bochum (RUB)

VI. Memory Management

May 17, 2023 (Summer Term 2023)



RUHR  
UNIVERSITÄT  
BOCHUM

RUB

[www.informatik.rub.de](http://www.informatik.rub.de)

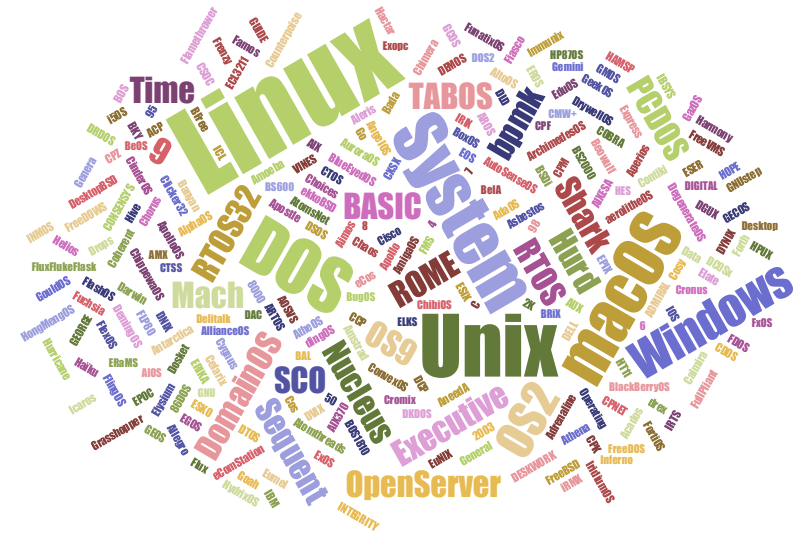
Chair of Operating Systems and System Software





# Agenda

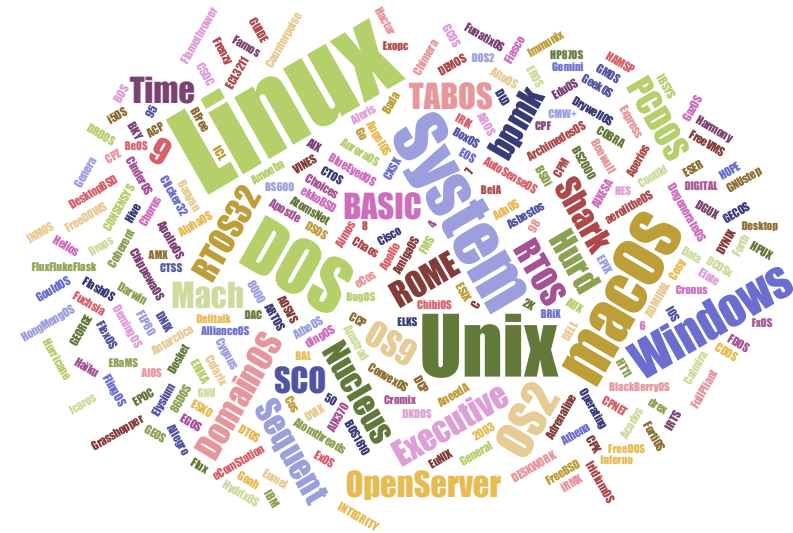
- ▶ Recap
- ▶ Organizational Matters
- ▶ Principles of Memory Management
  - ▶ Motivation
  - ▶ Requirements
  - ▶ Strategies
- ▶ Memory Allocation Schemes
- ▶ Memory Management for Multi-Program Operation
  - ▶ Segmentation
  - ▶ Paging
- ▶ Summary and Outlook





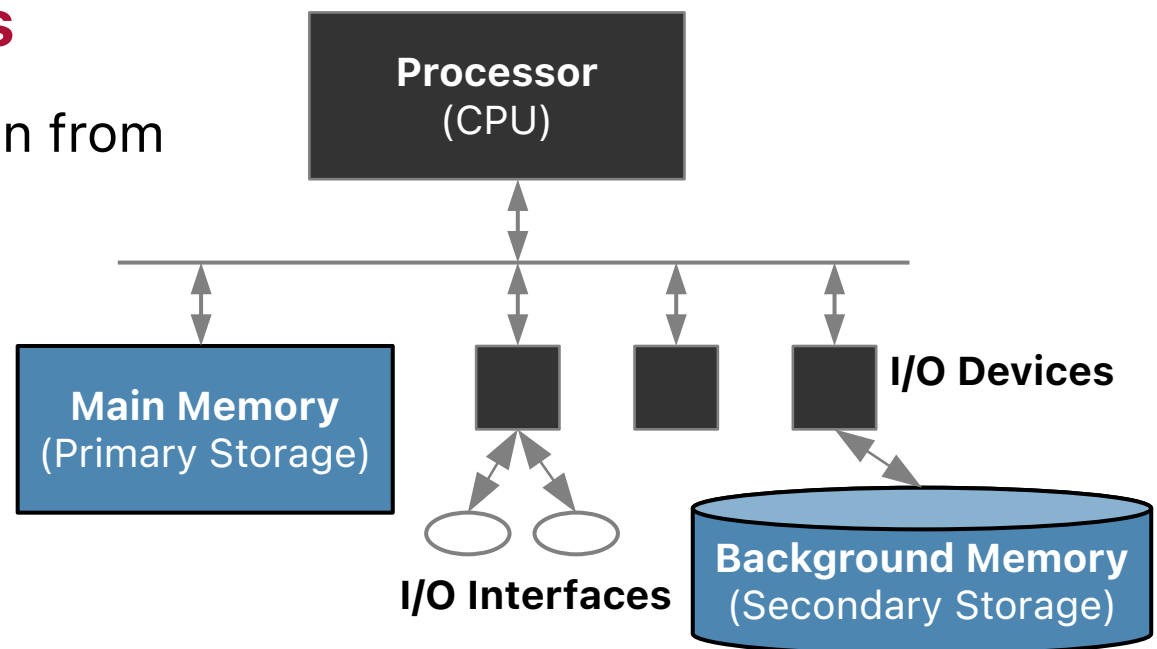
# Agenda

- ▶ Recap
- ▶ Organizational Matters
- ▶ Principles of Memory Management
  - ▶ Motivation
  - ▶ Requirements
  - ▶ Strategies
- ▶ Memory Allocation Schemes
- ▶ Memory Management for Multi-Program Operation
  - ▶ Segmentation
  - ▶ Paging
- ▶ Summary and Outlook



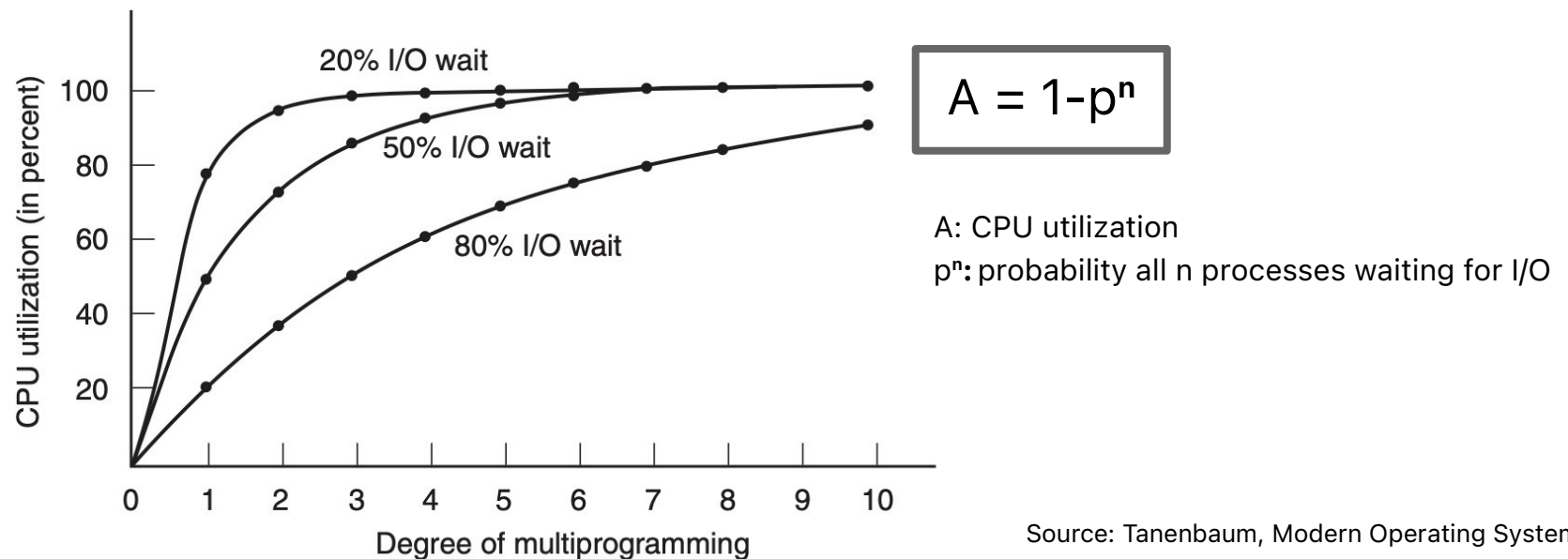
# Operating Resources (Recap)

- the operating system has the following responsibilities:
  - management of the computers' operating resources
  - **creation of abstractions** that allow applications to handle operating resources (more) easily and efficiently
- up to now: **processes**
  - concept for abstraction from the real CPU
- now: **memory**
  - management of main and background memory



# Motivation: Memory for Multi-Program Operation

- CPU utilisation assuming a certain I/O wait probability:

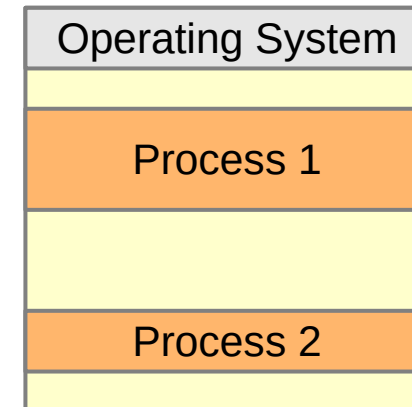


- ➔ **multi-program operation is essential for a high utilisation of the CPU**
- upon starting and terminating the processes, memory must be dynamically allocated or deallocated!



# Requirements for Memory Management

- multiple processes require main memory
  - processes are located at different places in the main memory
  - need for protection of the operating system and the processes among each other
  - memory may not be sufficient not enough for all processes



- **know, manage and allocate free memory areas**
- **swap-in and swap-out of processes**
- **relocation of program code**
- **exploit hardware support (i.e., memory management unit)**

# Strategies for Memory Management

Fundamental strategies at each level of the memory hierarchy:

- **placement strategy** (dt. Platzierungsstrategie)

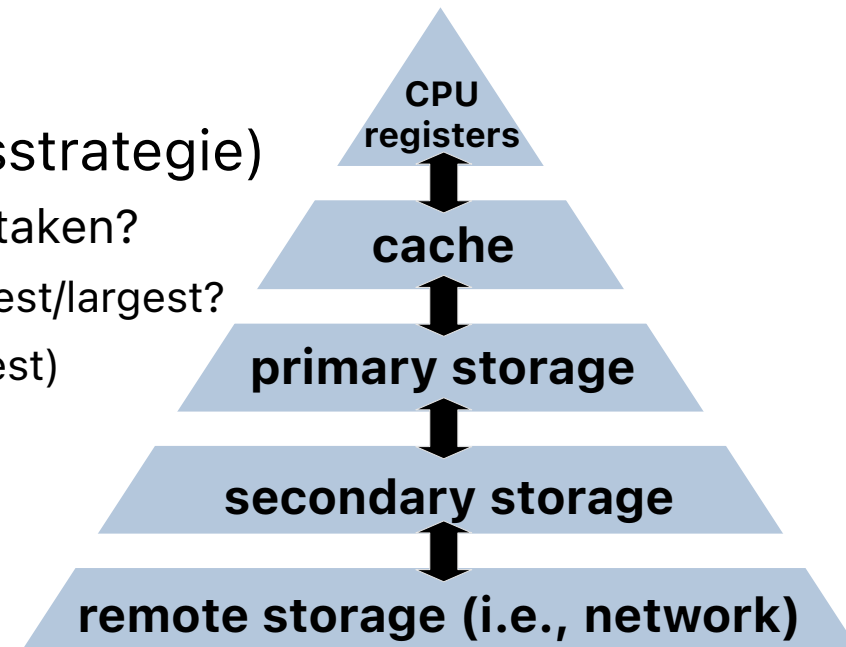
- **from where** should required memory be taken?
  - where is leftover (dt. Verschnitt) the smallest/largest?
  - do not care (leftover is of secondary interest)

- **fetch strategy** (dt. Ladestrategie)

- **when** should data be loaded to memory?
  - on demand
  - anticipatory (i.e., in advance)

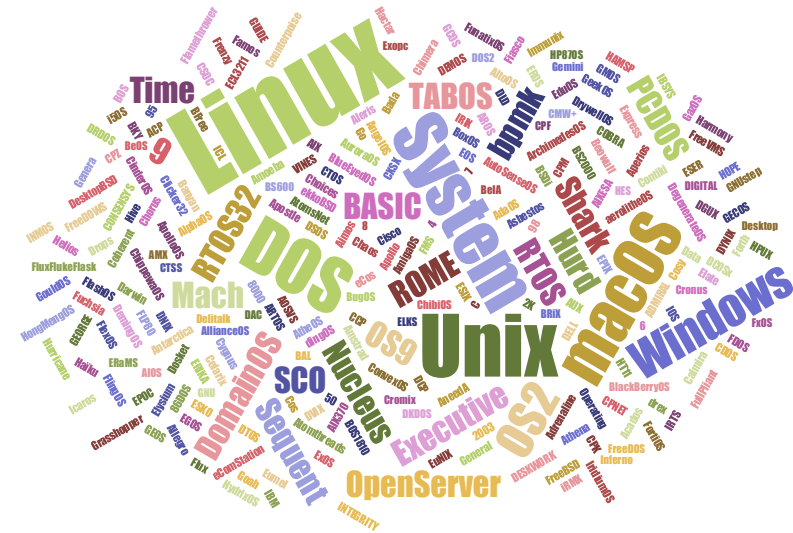
- **replacement strategy** (dt. Ersetzungsstrategie)

- **which** memory contents are to be replaced, if any, if the memory is running low?
  - the oldest, most rarely used memory
  - the longest unused memory



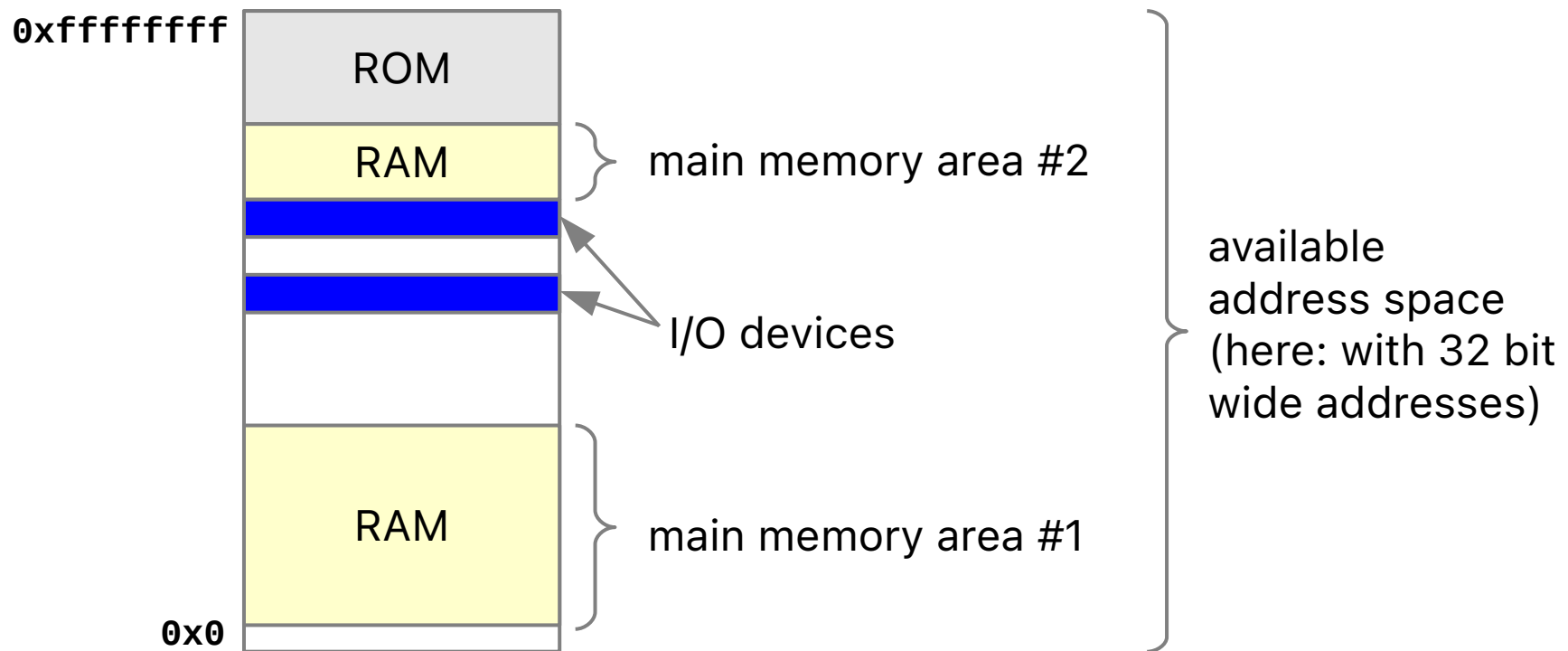
# Agenda

- ▶ Recap
- ▶ Organizational Matters
- ▶ Principles of Memory Management
  - ▶ Motivation
  - ▶ Requirements
  - ▶ Strategies
- ▶ **Memory Allocation Schemes**
- ▶ Memory Management for Multi-Program Operation
  - ▶ Segmentation
  - ▶ Paging
- ▶ Summary and Outlook



# Memory Allocation - Problem Statement

- available memory is non-contiguous and has holes



**memory map** of a (hypothetical) 32 bit system

# Static Memory Allocation

- fixed memory areas for operating system and user programs
- **problems**
  - degree of multiprogram operation limited
  - limitation of other resources (e.g., bandwidth for input/output due to too small buffers)
  - unused memory of the operating system cannot be used by application programs and vice versa

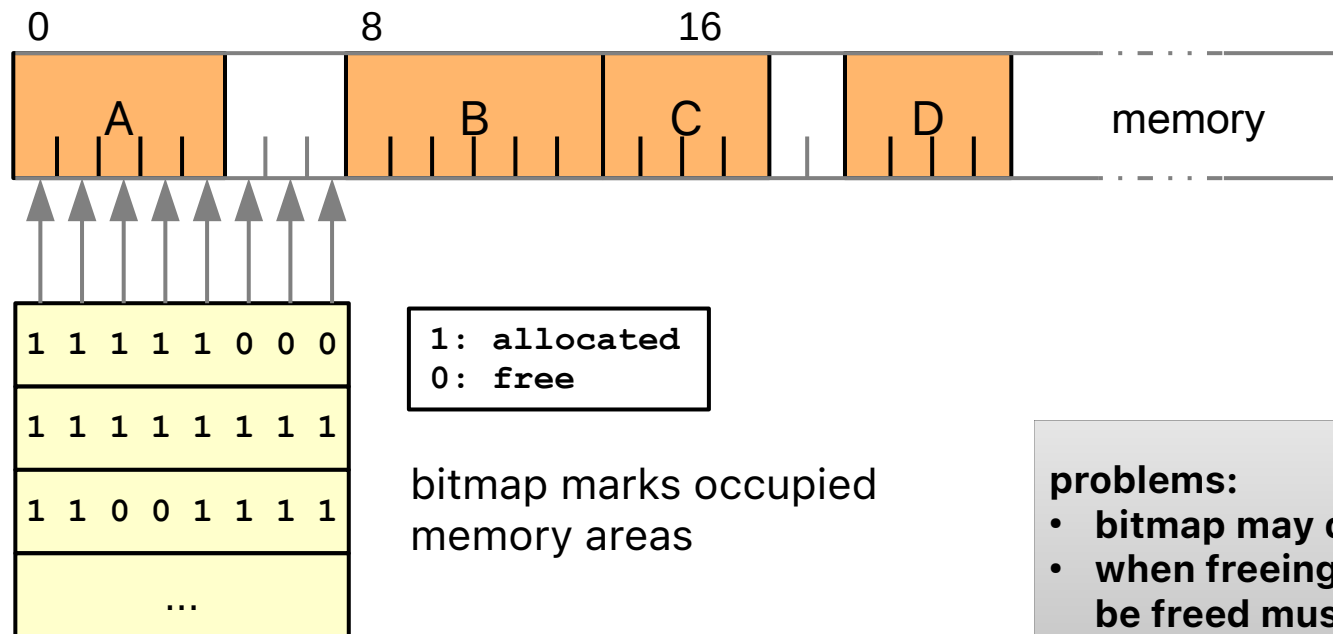
→ **dynamic memory management is mandatory**

# Dynamic Memory Allocation

- **segments**
  - **contiguous memory** area (i.e., memory range with consecutive addresses)
  - **variable size**
- **allocation** (dt. Belegung) und **deallocation** (dt. Freigabe) of memory segments
- an application program usually has the following segments:
  - text segment
  - data segment
  - stack segment (local variables, parameters, return addresses, ...)
- search for suitable memory areas for allocation
  - especially at program start
- ➔ **placement strategies are necessary**
  - especially important:  
efficient free-space management (dt. Freispeicherverwaltung)

# Free-Space Management – Bitmap

- free (possibly also allocated) segments of the memory must be represented
- **bitmap (or bit vector)**



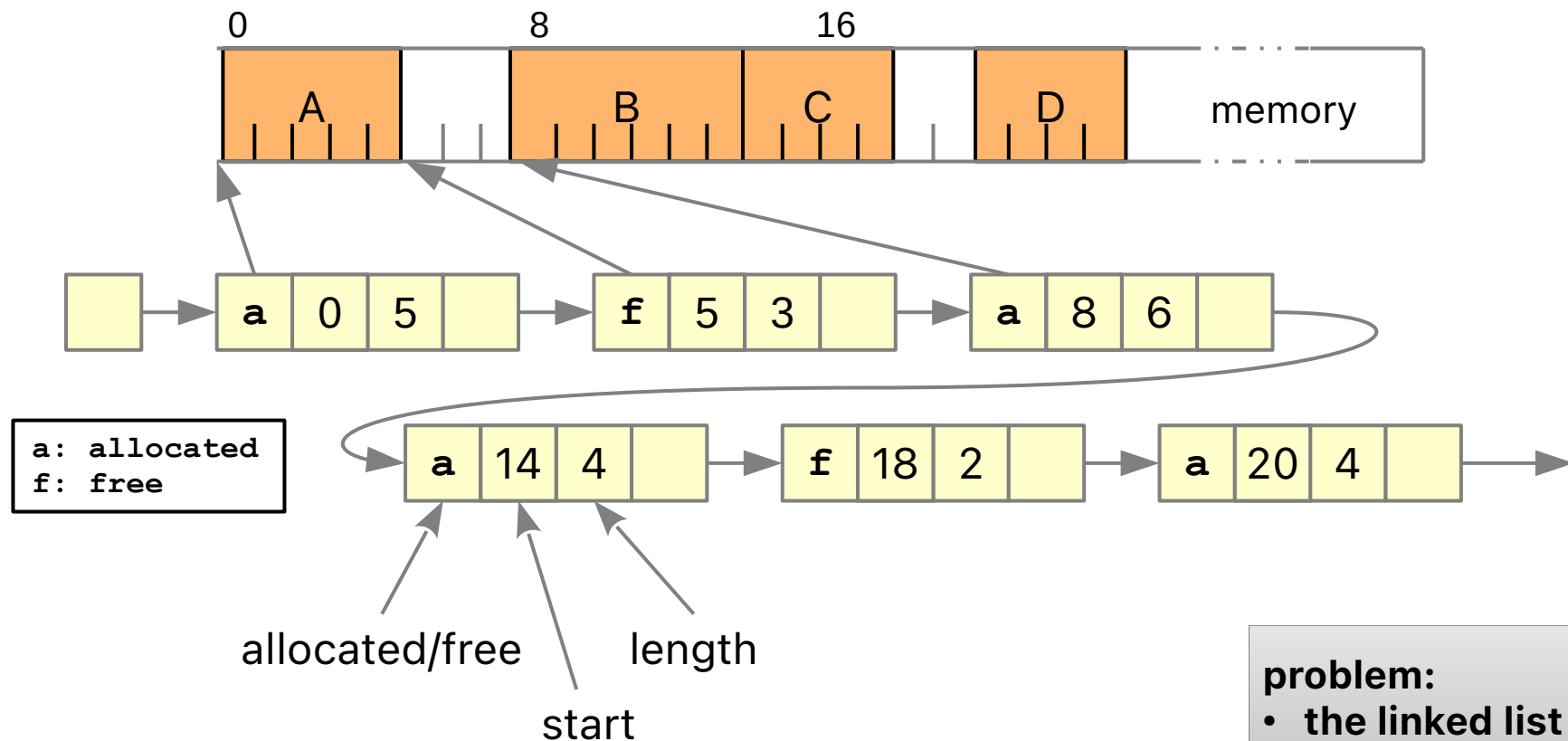
memory units of the same size (e.g., 1 byte, 64 bytes, 1024 bytes)

## problems:

- **bitmap may cost a lot of memory**
- **when freeing, size of the memory to be freed must be known/specified**
- **linear search necessary (→ runtime overhead)**
- **handling of high memory pressure**

# Free-Space Management – Linked List (V1)

## ■ linked list



### problem:

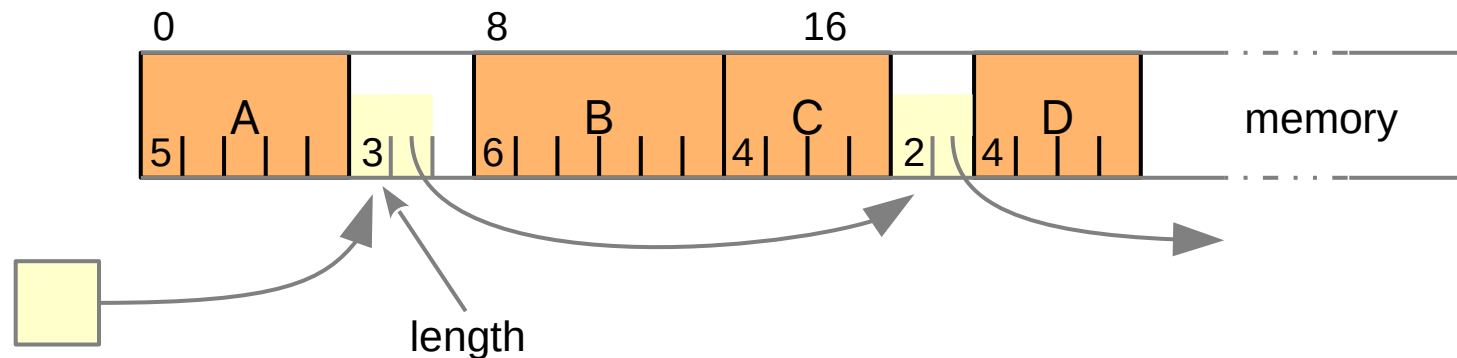
- the linked list itself requires (dynamically allocated) memory

representation of allocated (used) and free (unused) segments



# Free-Space Management – Linked List (V2)

- **keep linked list in unused (i.e., free) memory**

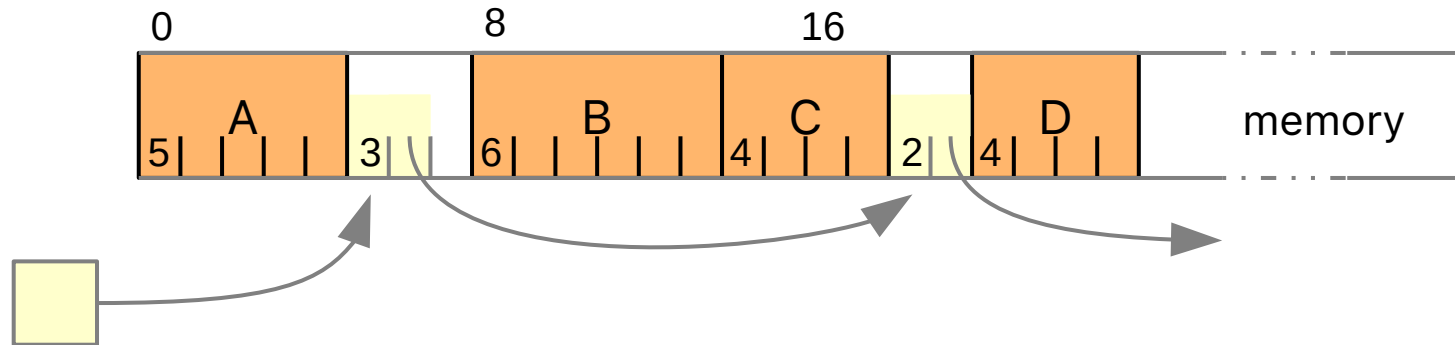


minimum hole size must be guaranteed

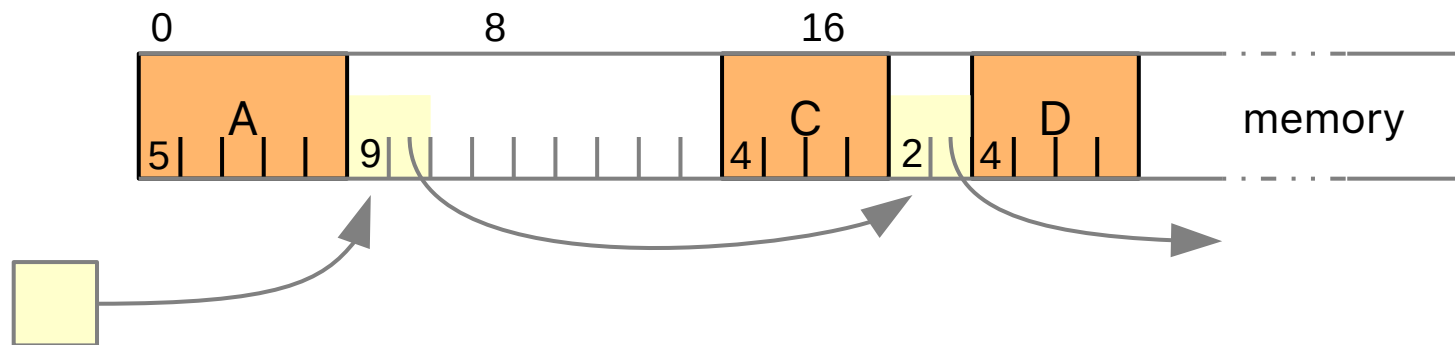
- backward linking may be necessary to increase efficiency
- representation ultimately also dependent on the allocation strategy

# Free-Space Management - Freeing Memory

- merging leftover holes



after freeing of B



# Placement Strategies – List-based Strategies

depending on the individual strategy, hole lists are sorted differently:

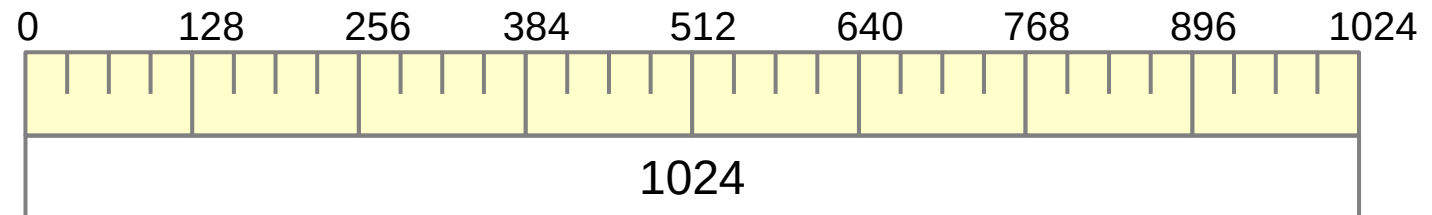
- **First Fit** (sorting: **by memory address**)
  - linear search, first matching hole is used
- **Rotating First Fit / Next Fit** (sorting: **by memory address**)
  - like First Fit, but start at the last assigned hole
  - avoids many small holes at the beginning of the list (like First Fit)
- **Best Fit** (sorting: **by hole size** – smallest hole first)
  - linear search, smallest matching hole is searched
- **Worst Fit** (sorting: **by hole size** – largest hole first)
  - largest matching hole is searched
- **common problem:**
  - when holes are too small → memory fragmentation

# Placement Strategies – Buddy System

- memory allocation algorithm: buddy system
- operating principle
  - allocation: split available memory into **partitions of size  $2^n$  bytes**
  - try to fit memory request:
    - splitting memory into halves
    - best fit
  - free: recursively merge any free buddy partitions
- practical importance
  - Linux (Buddy System variant which addresses external fragmentation)
  - FreeBSD (jemalloc, which implements the Buddy System)

# Placement Strategies – Buddy System

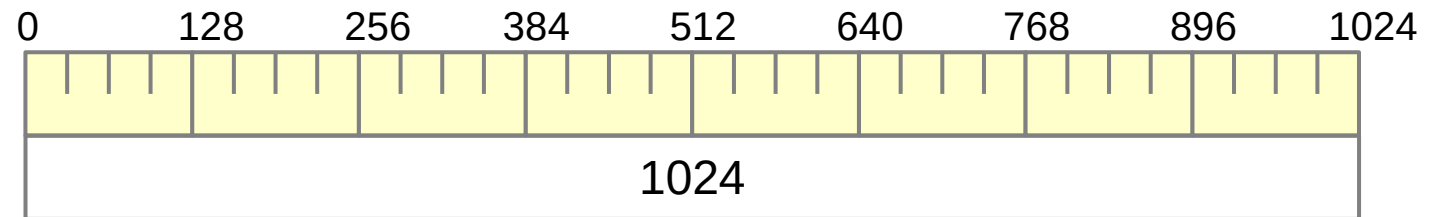
- the **buddy system** partitions memory into dynamic areas of size  $2^n$



- A: request 70
- B: request 35
- C: request 80  
release A
- D: request 60  
release B  
release D  
release C

# Placement Strategies – Buddy System

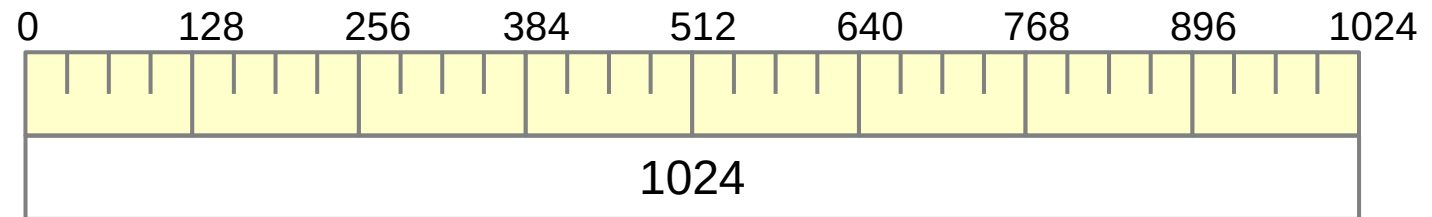
- the **buddy system** partitions memory into dynamic areas of size  $2^n$



		<u>bucket size</u>	<u>free list</u>	<u>allocated list</u>
A:	→ request 70			
B:	request 35	1024 ( $2^{10}$ )	0	
C:	request 80	512 ( $2^9$ )		
	release A	256 ( $2^8$ )		
D:	request 60	128 ( $2^7$ )		
	release B	64 ( $2^6$ )		
	release D			
	release C			

# Placement Strategies – Buddy System

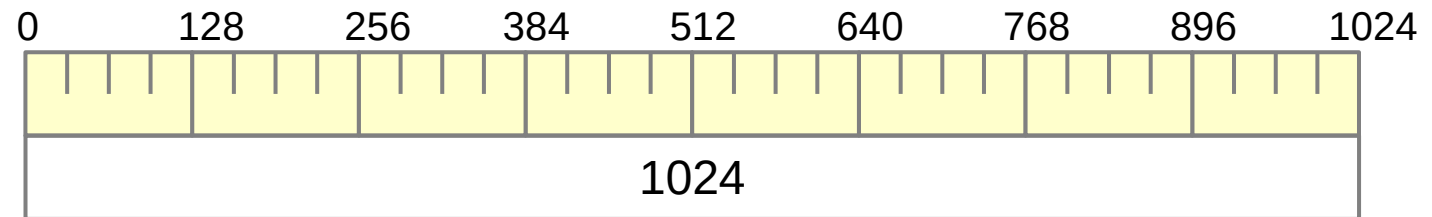
- the **buddy system** partitions memory into dynamic areas of size  $2^n$



		<u>bucket size</u>	<u>free list</u>	<u>allocated list</u>
A:	→ request 70			
B:	request 35	1024 ( $2^{10}$ )		
C:	request 80	512 ( $2^9$ )	0, 512	
	release A	256 ( $2^8$ )		
D:	request 60	128 ( $2^7$ )		
	release B	64 ( $2^6$ )		
	release D			
	release C			

# Placement Strategies – Buddy System

- the **buddy system** partitions memory into dynamic areas of size  $2^n$

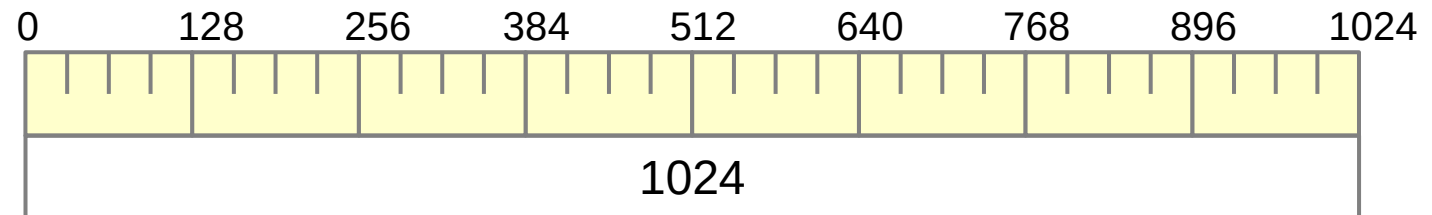


	<u>bucket size</u>	<u>free list</u>	<u>allocated list</u>
A: → request 70			
B: request 35	1024 ( $2^{10}$ )		
C: request 80	512 ( $2^9$ )	512	
release A	256 ( $2^8$ )	0, 256	
D: request 60	128 ( $2^7$ )		
release B	64 ( $2^6$ )		
release D			
release C			



# Placement Strategies – Buddy System

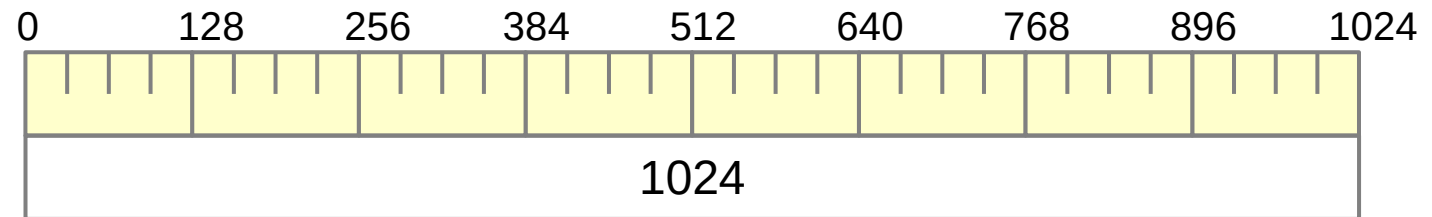
- the **buddy system** partitions memory into dynamic areas of size  $2^n$



		<u>bucket size</u>	<u>free list</u>	<u>allocated list</u>
A:	→ request 70			
B:	request 35	1024 ( $2^{10}$ )		
C:	request 80	512 ( $2^9$ )	512	
	release A	256 ( $2^8$ )	256	
D:	request 60	128 ( $2^7$ )	0, 128	→ allocate A at 0
	release B	64 ( $2^6$ )		
	release D			
	release C			

# Placement Strategies – Buddy System

- the **buddy system** partitions memory into dynamic areas of size  $2^n$



A: → request 70

B: request 35

C: request 80

release A

D: request 60

release B

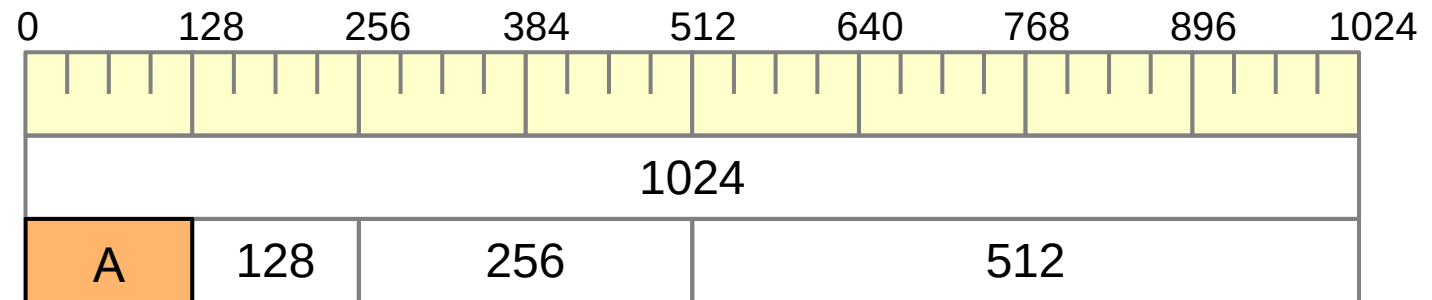
release D

release C

<u>bucket size</u>	<u>free list</u>	<u>allocated list</u>
1024 ( $2^{10}$ )		address → size
512 ( $2^9$ )	512	A: 0 (128)
256 ( $2^8$ )	256	
128 ( $2^7$ )	128	
64 ( $2^6$ )		

# Placement Strategies – Buddy System

- the **buddy system** partitions memory into dynamic areas of size  $2^n$



A: request 70

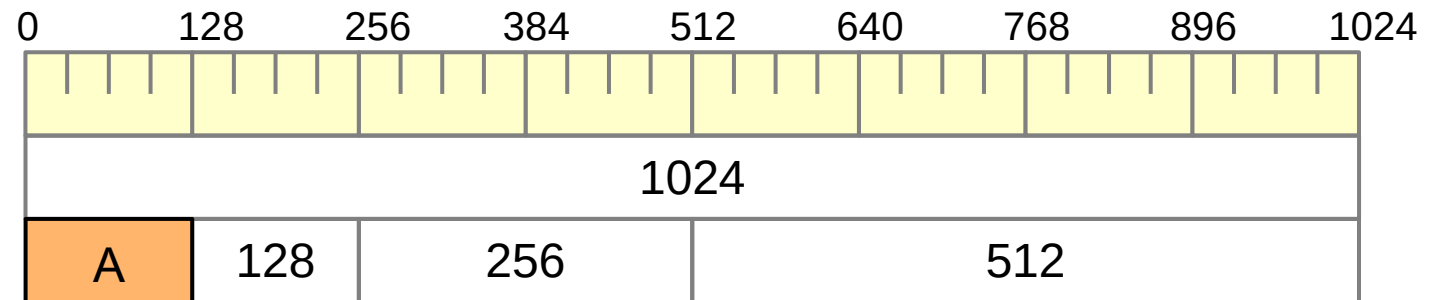
B: → request 35

C: request 80  
release A

D: request 60  
release B  
release D  
release C

# Placement Strategies – Buddy System

- the **buddy system** partitions memory into dynamic areas of size  $2^n$



A: request 70

B: → request 35

C: request 80

release A

D: request 60

release B

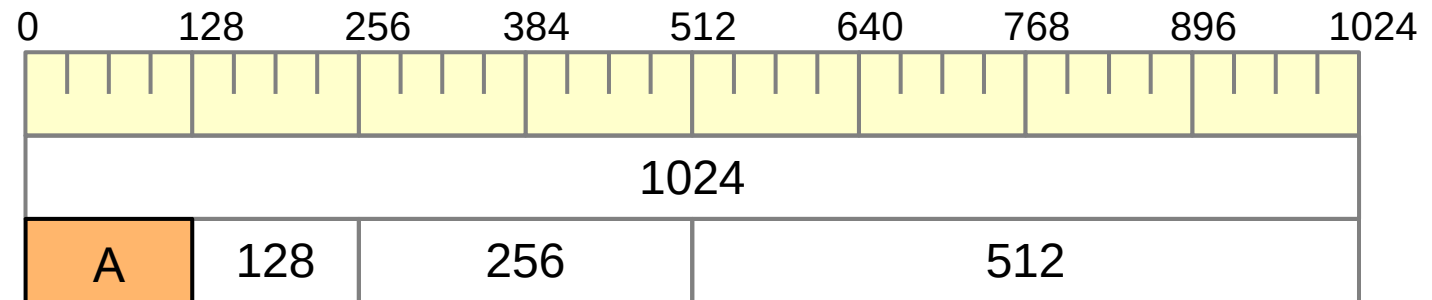
release D

release C

<u>bucket size</u>	<u>free list</u>	<u>allocated list</u>
1024 ( $2^{10}$ )		
512 ( $2^9$ )	512	A: 0 (128)
256 ( $2^8$ )	256	
128 ( $2^7$ )	128	
64 ( $2^6$ )		

# Placement Strategies – Buddy System

- the **buddy system** partitions memory into dynamic areas of size  $2^n$



A: request 70

B: → request 35

C: request 80

release A

D: request 60

release B

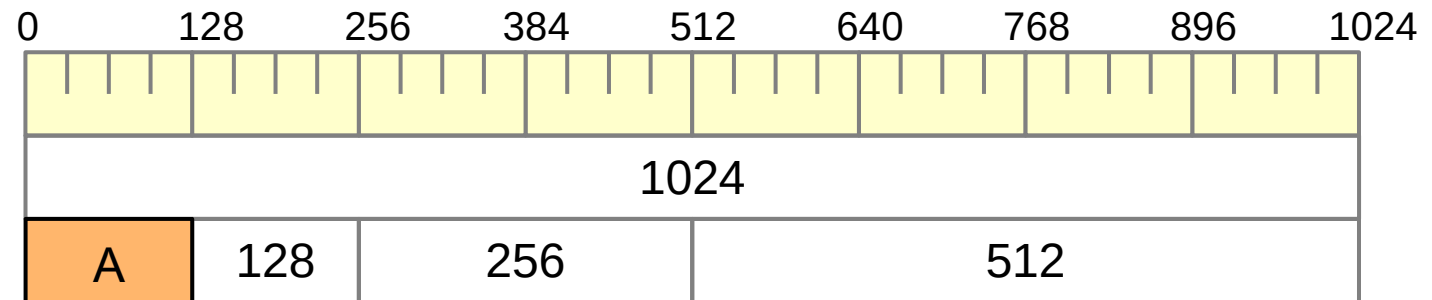
release D

release C

<u>bucket size</u>	<u>free list</u>	<u>allocated list</u>
1024 ( $2^{10}$ )		A: 0 (128)
512 ( $2^9$ )	512	
256 ( $2^8$ )	256	
128 ( $2^7$ )		
64 ( $2^6$ )	128, 192	→ allocate B at 128

# Placement Strategies – Buddy System

- the **buddy system** partitions memory into dynamic areas of size  $2^n$



A: request 70

B: → request 35

C: request 80

release A

D: request 60

release B

release D

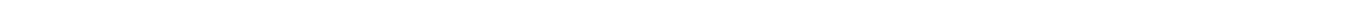
release C

bucket sizefree listallocated list1024 ( $2^{10}$ )512 ( $2^9$ ) 512256 ( $2^8$ ) 256128 ( $2^7$ )64 ( $2^6$ ) 192

A: 0 (128)

B: 128 (64)

# Placement Strategies – Buddy System

- the **buddy system** partitions memory into dynamic areas of size  $2^n$
- 

1024			
A	128	256	512
A	B 64	256	512

~~A: request 70~~

~~B: request 35~~

C: → request 80

release A

D: request 60

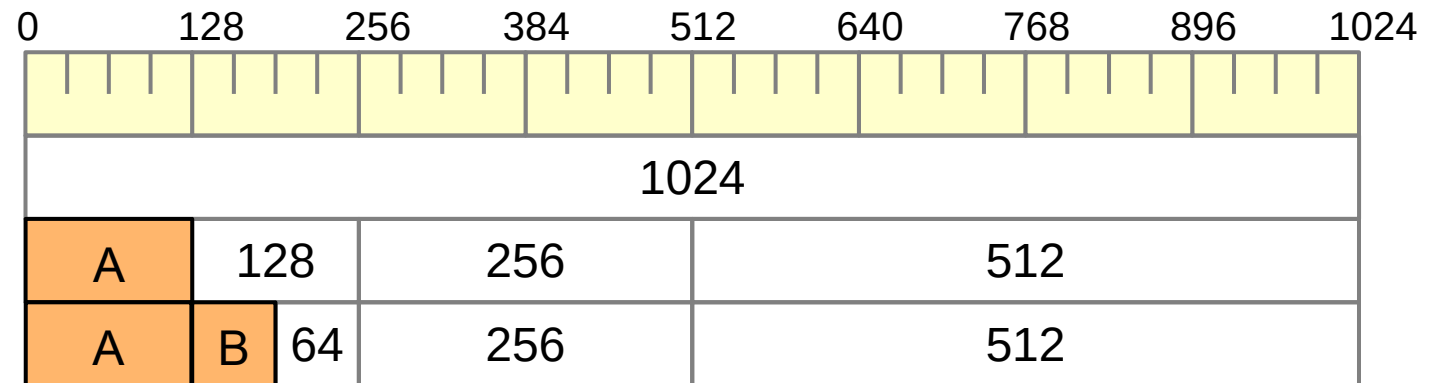
release B

release D

release C

# Placement Strategies – Buddy System

- the **buddy system** partitions memory into dynamic areas of size  $2^n$



C: → request 80

release A

D: request 60

release B

release D

release C

bucket sizefree listallocated list1024 ( $2^{10}$ )512 ( $2^9$ )256 ( $2^8$ )128 ( $2^7$ )64 ( $2^6$ )

512

256

192

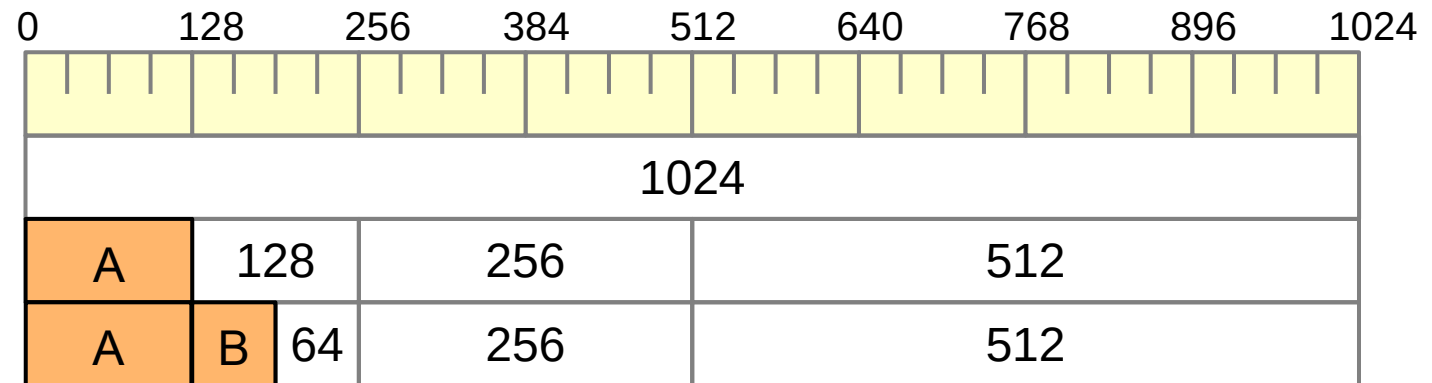
A: 0 (128)

B: 128 (64)



# Placement Strategies – Buddy System

- the **buddy system** partitions memory into dynamic areas of size  $2^n$



C: → request 80

release A

D: request 60

release B

release D

release C

bucket sizefree listallocated list1024 ( $2^{10}$ )512 ( $2^9$ )256 ( $2^8$ )128 ( $2^7$ )64 ( $2^6$ )

512

256, 384

192

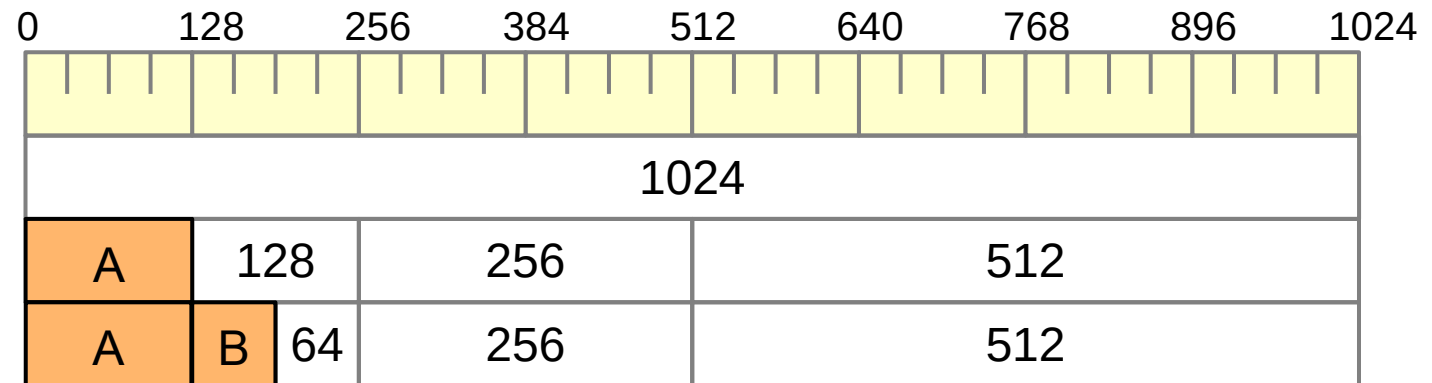
A: 0 (128)

B: 128 (64)

→ allocate C at 256

# Placement Strategies – Buddy System

- the **buddy system** partitions memory into dynamic areas of size  $2^n$



C: → request 80

release A

D: request 60

release B

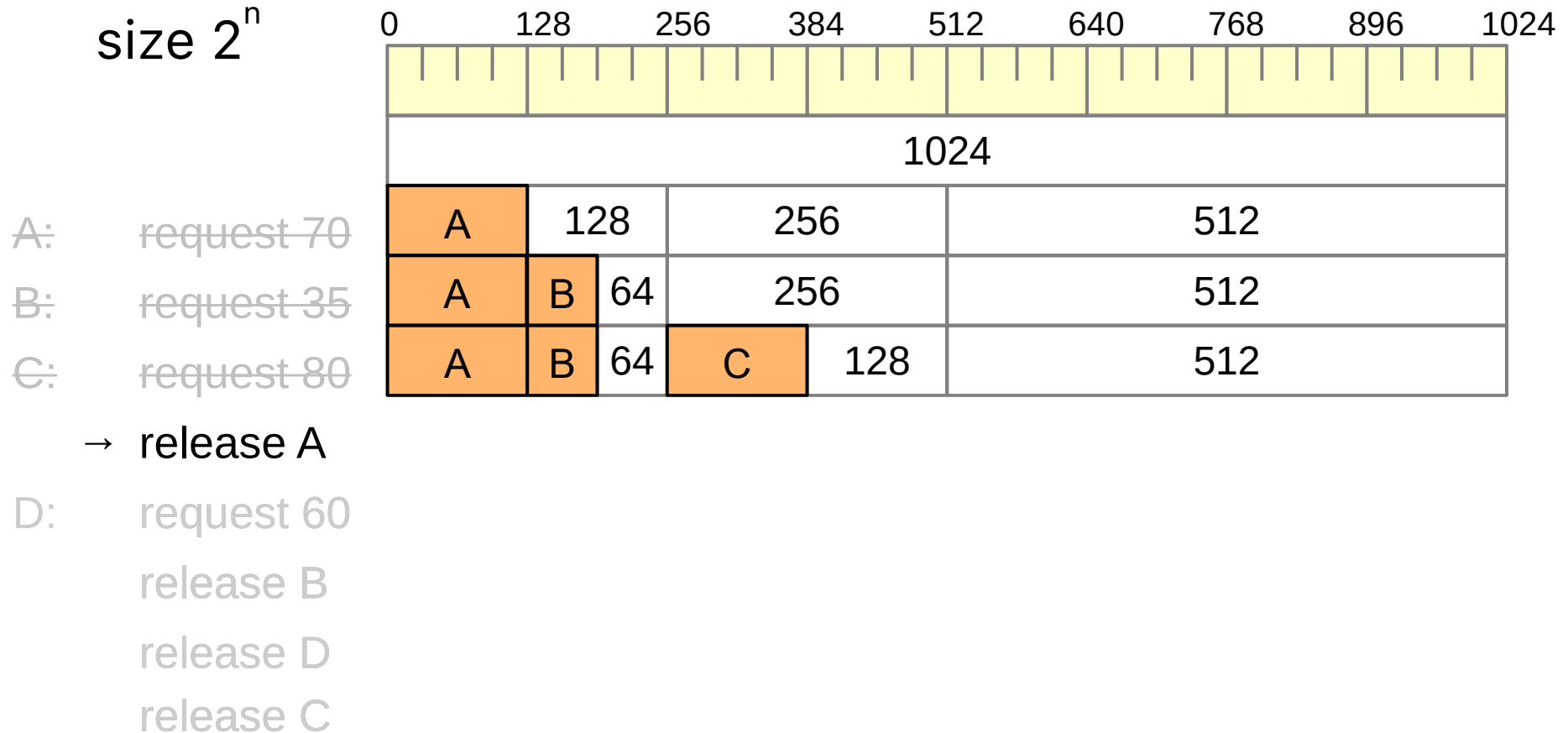
release D

release C

<u>bucket size</u>	<u>free list</u>	<u>allocated list</u>
1024 ( $2^{10}$ )		A: 0 (128)
512 ( $2^9$ )	512	B: 128 (64)
256 ( $2^8$ )		C: 256 (128)
128 ( $2^7$ )	384	
64 ( $2^6$ )	192	

# Placement Strategies – Buddy System

- the **buddy system** partitions memory into dynamic areas of size  $2^n$



# Placement Strategies – Buddy System

- the **buddy system** partitions memory into dynamic areas of size  $2^n$

		0	128	256	384	512	640	768	896	1024
		1024								
A: request 70	A	128	256	512						
B: request 35	A	B 64	256	512						
C: request 80	A	B 64	C 128	512						
→ release A										
D: request 60										
release B										
release D										
release C										

	<u>bucket</u>	<u>size</u>	<u>free list</u>	<u>allocated list</u>
	1024	( $2^{10}$ )		A: 0 (128)
	512	( $2^9$ )	512	B: 128 (64)
	256	( $2^8$ )		C: 256 (128)
	128	( $2^7$ )	384	
	64	( $2^6$ )	192	

# Placement Strategies – Buddy System

- the **buddy system** partitions memory into dynamic areas of size  $2^n$

		0	128	256	384	512	640	768	896	1024
		1024								
A: request 70	A	128	256	512						
B: request 35	A	B 64	256	512						
C: request 80	128	B 64	C 128	512						
→ release A										
D: request 60										
release B										
release D										
release C										

	<u>bucket size</u>	<u>free list</u>	<u>allocated list</u>
	1024 ( $2^{10}$ )		<del>A: 0 (128)</del>
	512 ( $2^9$ )	512	B: 128 (64)
	256 ( $2^8$ )		C: 256 (128)
	128 ( $2^7$ )	0, 384	
	64 ( $2^6$ )	192	

# Placement Strategies – Buddy System

- the **buddy system** partitions memory into dynamic areas of size  $2^n$

		0	128	256	384	512	640	768	896	1024
		1024								
A: request 70	A	128	256		512					
B: request 35	A	B	64	256		512				
C: request 80	A	B	64	C	128	512				
release A	128	B	64	C	128	512				
D: → request 60										
release B										
release D										
release C										

# Placement Strategies – Buddy System

- the **buddy system** partitions memory into dynamic areas of size  $2^n$

size 2<sup>n</sup>

01282563845126407688961024

1024

A128256512

A B 64256512

A B 64 C 128512

128 B 64 C 128512

A: request 70

B: request 35

C: request 80

release A

D: → request 60

release B

release D

release C

bucket size

free list

allocated list

1024 (2<sup>10</sup>)

512 (2<sup>9</sup>)

256 (2<sup>8</sup>)

128 (2<sup>7</sup>)

64 (2<sup>6</sup>)

512

0, 384

192

A: 0 (128)

B: 128 (64)

C: 256 (128)

→ allocate D at 192

# Placement Strategies – Buddy System

- the **buddy system** partitions memory into dynamic areas of size  $2^n$

size 2<sup>n</sup>

01282563845126407688961024

1024

A128256512

A64256512

A64128512

12864128512

A: request 70

B: request 35

C: request 80

release A

D: → request 60

release B

release D

release C

bucket size

free list

allocated list

1024 (2<sup>10</sup>)

512 (2<sup>9</sup>)

256 (2<sup>8</sup>)

128 (2<sup>7</sup>)

64 (2<sup>6</sup>)

512

0, 384

A: 0 (128)

B: 128 (64)

C: 256 (128)

D: 192 (64)



# Placement Strategies – Buddy System

- the **buddy system** partitions memory into dynamic areas of size  $2^n$

		0	128	256	384	512	640	768	896	1024
		1024								
A: request 70	A	128	256		512					
B: request 35	A	B	64	256		512				
C: request 80	A	B	64	C	128	512				
release A	128	B	64	C	128	512				
D: request 60	128	B	D	C	128	512				

→ release B

release D

release C

- freeing the memory allocated for request B
- buddy (D) remains allocated
- no merging of leftover holes

# Placement Strategies – Buddy System

- the **buddy system** partitions memory into dynamic areas of size  $2^n$

		0	128	256	384	512	640	768	896	1024
		1024								
A: request 70	A	128	256		512					
B: request 35	A	B	64	256		512				
C: request 80	A	B	64	C	128	512				
release A	128	B	64	C	128	512				
D: request 60	128	B	D	C	128	512				
→ release B	128	64	D	C	128	512				
release D										
release C										

# Placement Strategies – Buddy System

- the **buddy system** partitions memory into dynamic areas of size  $2^n$

		0	128	256	384	512	640	768	896	1024
		1024								
A:	request 70	A	128	256	512					
B:	request 35	A	B 64	256	512					
C:	request 80	A	B 64	C 128	512					
	release A	128	B 64	C 128	512					
D:	request 60	128	B D	C 128	512					
	release B	128	64 D	C 128	512					

→ release D  
release C

- freeing the memory allocated for request D
- no buddy allocation present
- merge leftover holes, recursively (2 times)

# Placement Strategies – Buddy System

- the **buddy system** partitions memory into dynamic areas of size  $2^n$

size 2<sup>n</sup>

		0	128	256	384	512	640	768	896	1024
		1024								
A: request 70	A	128	256		512					
B: request 35	A	B	64	256		512				
C: request 80	A	B	64	C	128	512				
release A	128	B	64	C	128	512				
D: request 60	128	B	D	C	128	512				
release B	128	64	D	C	128	512				
release D	256			C	128	512				
→ release C	• freeing the memory allocated for request C									

- freeing the memory allocated for request C
- no buddy allocation present
- merge leftover holes, recursively (3 times)

# Placement Strategies – Buddy System

- the **buddy system** partitions memory into dynamic areas of size  $2^n$

size 2<sup>n</sup>

		0	128	256	384	512	640	768	896	1024
		1024								
A:	request 70	A	128	256		512				
B:	request 35	A	B	64	256		512			
C:	request 80	A	B	64	C	128	512			
	release A	128	B	64	C	128	512			
D:	request 60	128	B	D	C	128	512			
	release B	128	64	D	C	128	512			
	release D	256			C	128	512			
	release C	1024								

# Discussion – Fragmentation

## ■ external fragmentation

- **outside** the allocated memory area, memory fragments are created that can no longer be used
- occurs when using **list-based strategies** such as (first fit, best fit, etc.) and also buddy system when merge of buddies is not possible
- **countermeasures:** merge leftover holes, relocate memory areas

## ■ internal fragmentation

- there is unused memory **within** the allocated memory areas
- occurs when using the **Buddy system**, for example, as the requirements are rounded up to the next larger power of two
- undiscoverable, **invalid pointers** accessing fragmented areas
- **countermeasures:** difficult to address; alter granularity of memory allocation or different memory allocation scheme that better fits to the observed memory allocation patterns

# Active Use of the Discussed Methods

- use in the operating system
  - management of the system memory
  - allocation of memory to processes **and** the operating system
- use within a process
  - management of the heap memory
  - allows dynamic allocation of memory areas processes (malloc and free)
- use for secondary storage
  - management of specific sections of secondary storage, for example, memory areas for process swap data (swap space)

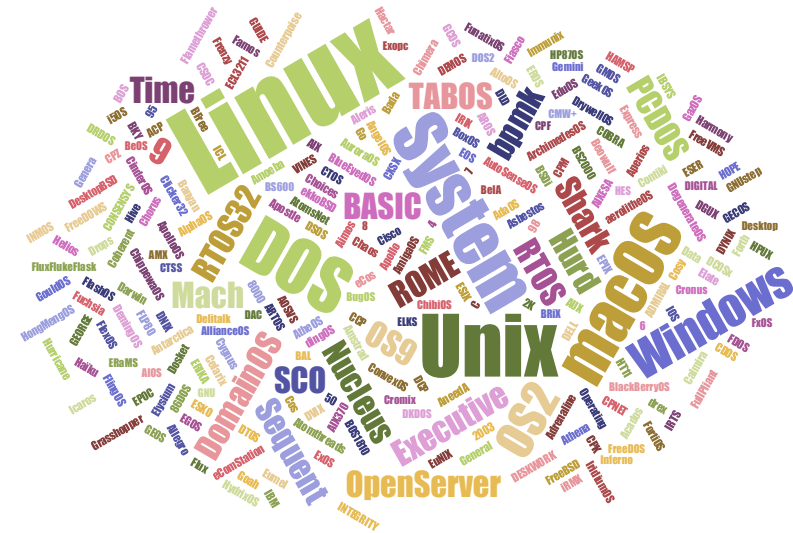
**Buddy  
allocator in  
Linux**

**typically list-  
based**

**often with  
implemented  
with bitmaps**

# Agenda

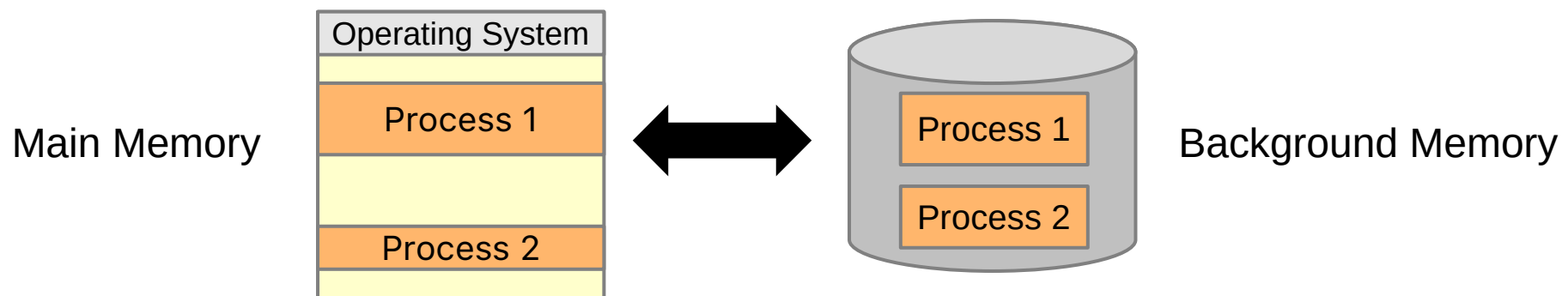
- ▶ Recap
- ▶ Organizational Matters
- ▶ Principles of Memory Management
  - ▶ Motivation
  - ▶ Requirements
  - ▶ Strategies
- ▶ Memory Allocation Schemes
- ▶ Memory Management for Multi-Program Operation
  - ▶ Segmentation
  - ▶ Paging
- ▶ Summary and Outlook





# Memory Management – Swapping

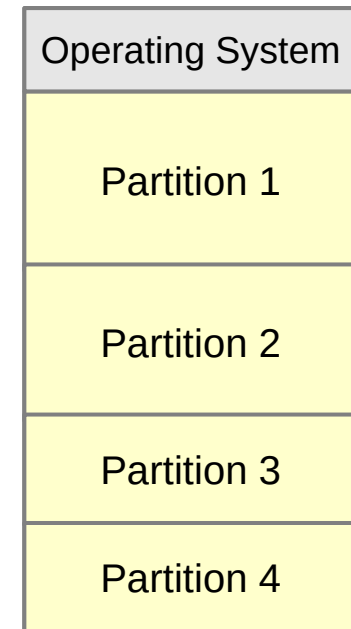
- segments of a process are swapped out to background memory and then freed in main memory
  - for example, to bridge waiting times for I/O
- swapping-in of the segments to the main memory at the end of the waiting time



- considerations: start/stop programs, swap-in/out processes
  - **when** (aligned to scheduling policy, consider overheads)?
  - **where** to place processes in main memory?

# Memory Management – Swapping

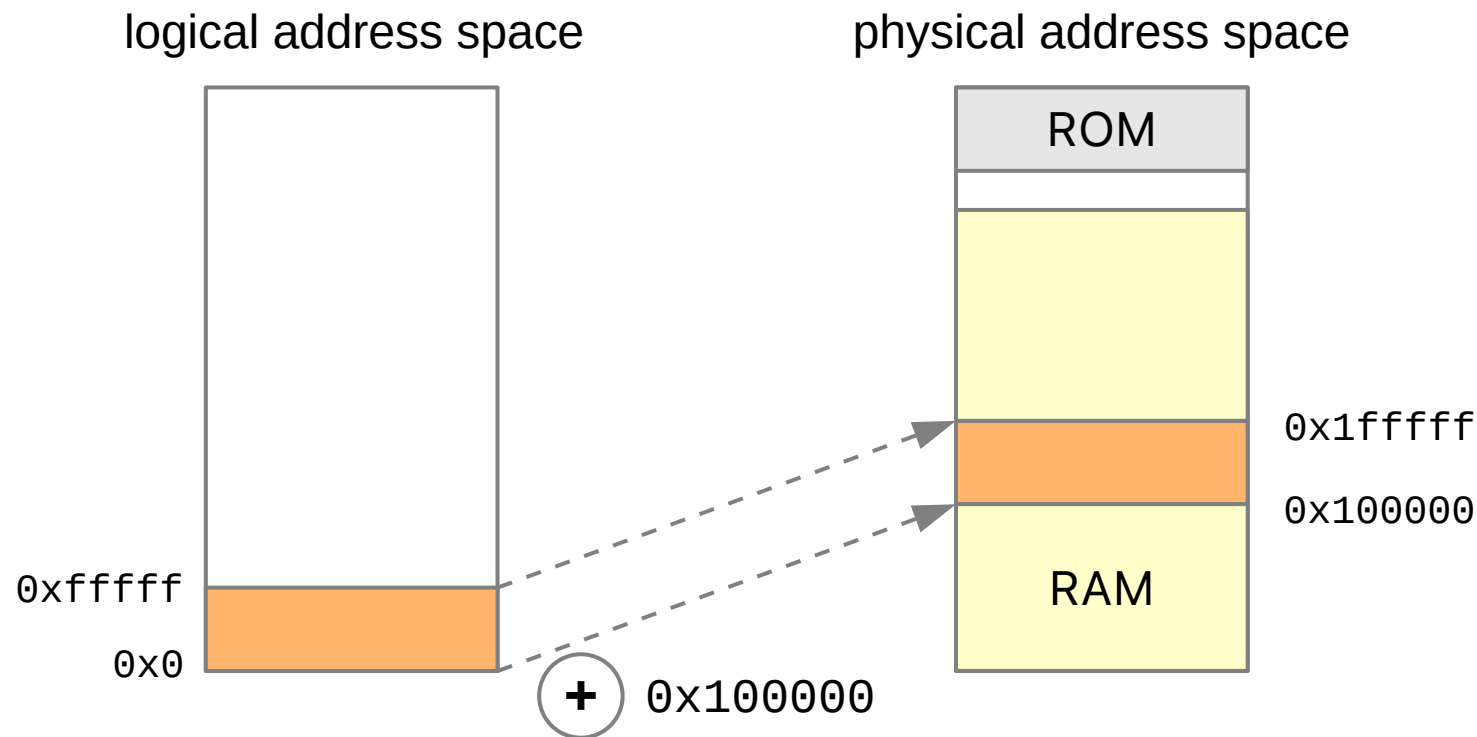
- addresses in the processes are usually statically bound
  - can only be loaded to the same place in the main memory
  - collisions with segments that may be new in the main memory
- possible solution 1: **static partitioning** of the main memory
  - only one process is running in each partition
  - swapped in processes are placed again into the same partition
  - memory can not be used optimally
- possible solution 2: **program relocation**
  - at load time (static linking)
  - during execution time (dynamic linking)



➔ focus: **dynamic memory allocation** with hardware support

# Memory Management – Segmentation

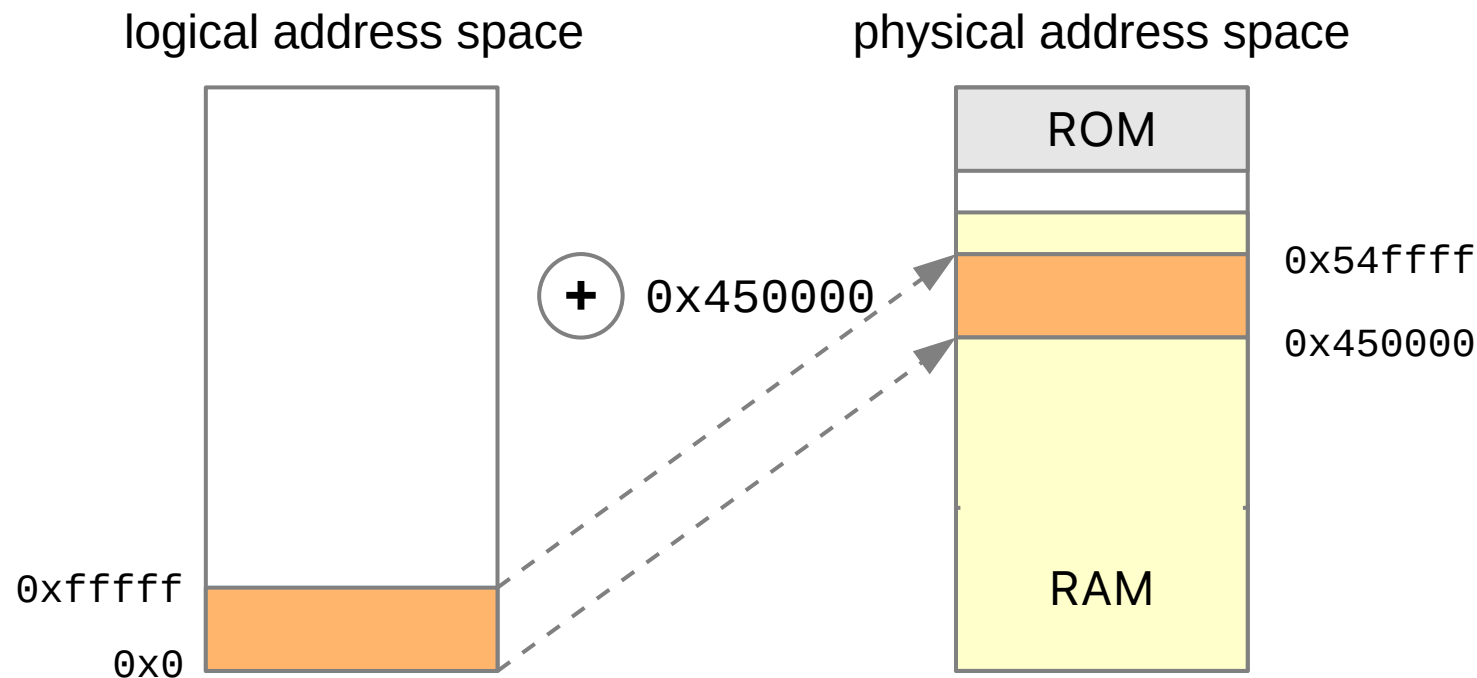
- map logical to physical addresses with hardware support



- a segment of the logical address space can be located at any position in the physical address space
- size of segments is variable**
- OS determines where a segment should actually be located in the physical address space

# Memory Management – Segmentation

- map logical to physical addresses with hardware support

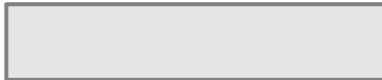


- a segment of the logical address space can be located at any position in the physical address space
- size of segments is variable**
- OS determines where a segment should actually be located in the physical address space

# Memory Management – Segmentation

- realisation with translation table (per process)

segment table base register



## segment table

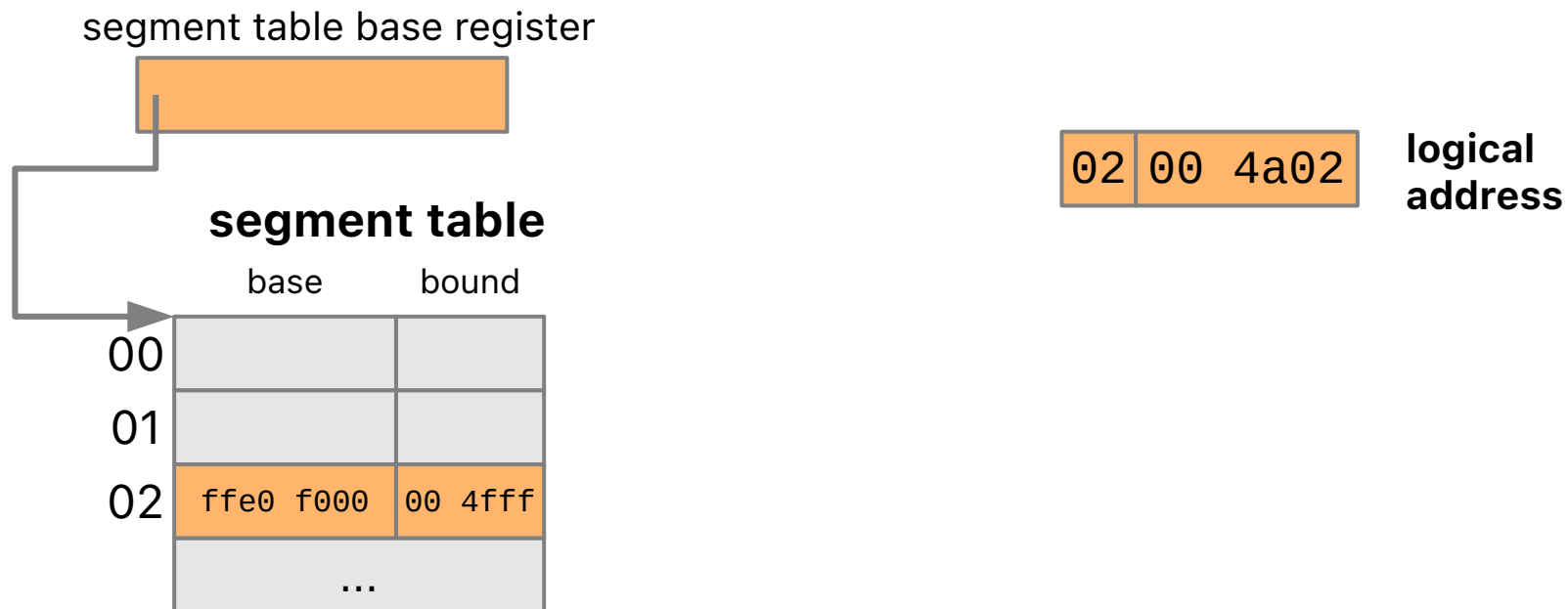
	base	bound
00		
01		
02		
	...	

02 00 4a02

logical  
address

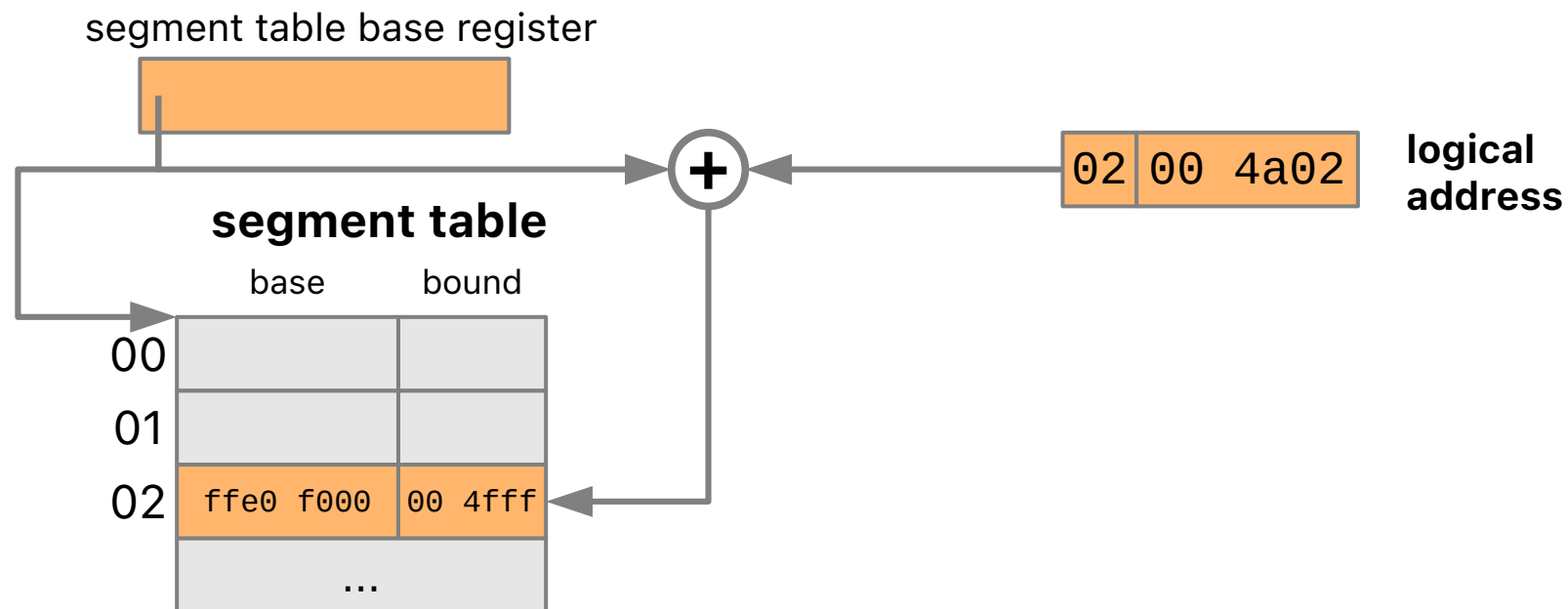
# Memory Management – Segmentation

- realisation with translation table (per process)



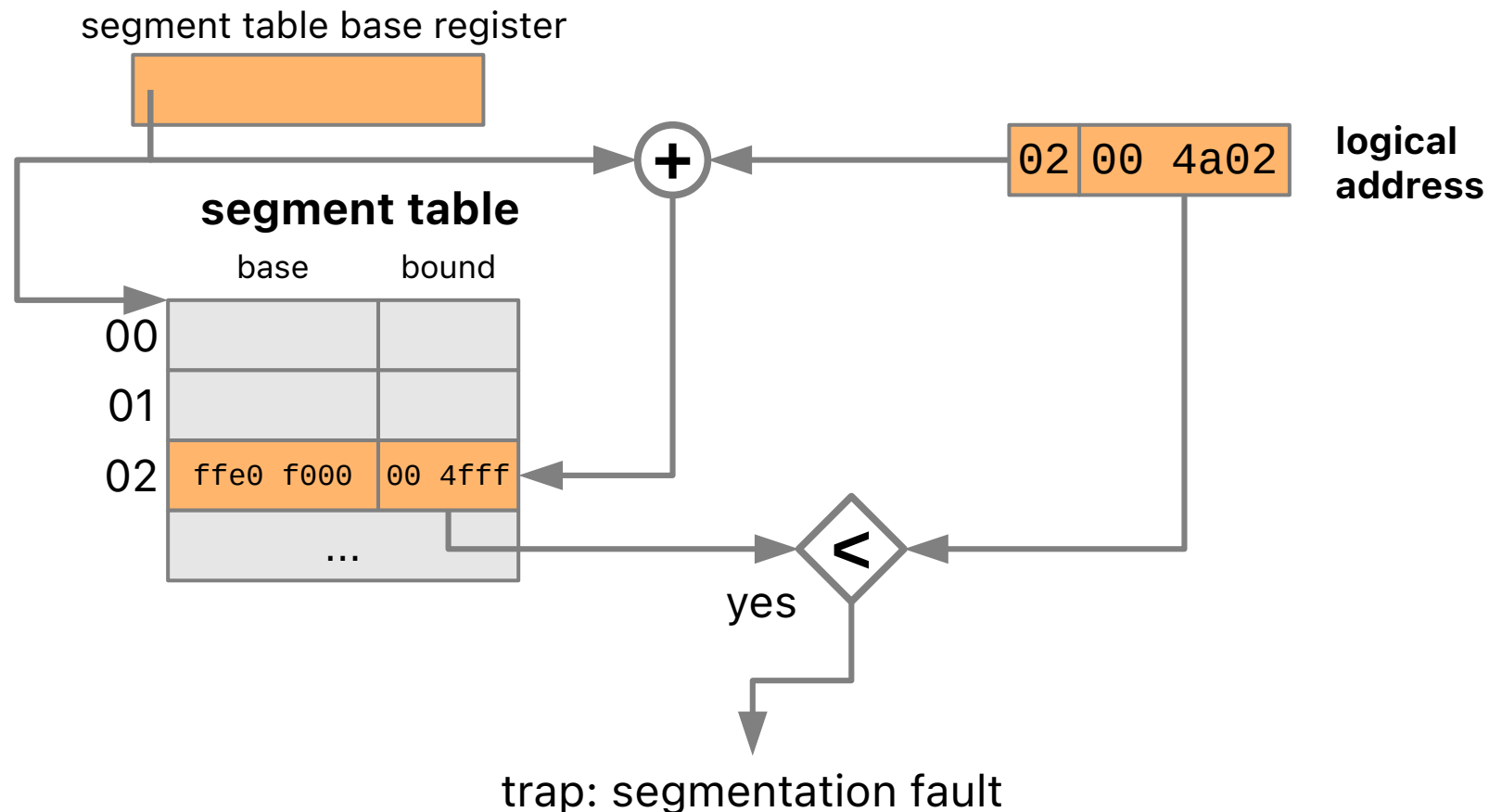
# Memory Management – Segmentation

- realisation with translation table (per process)



# Memory Management – Segmentation

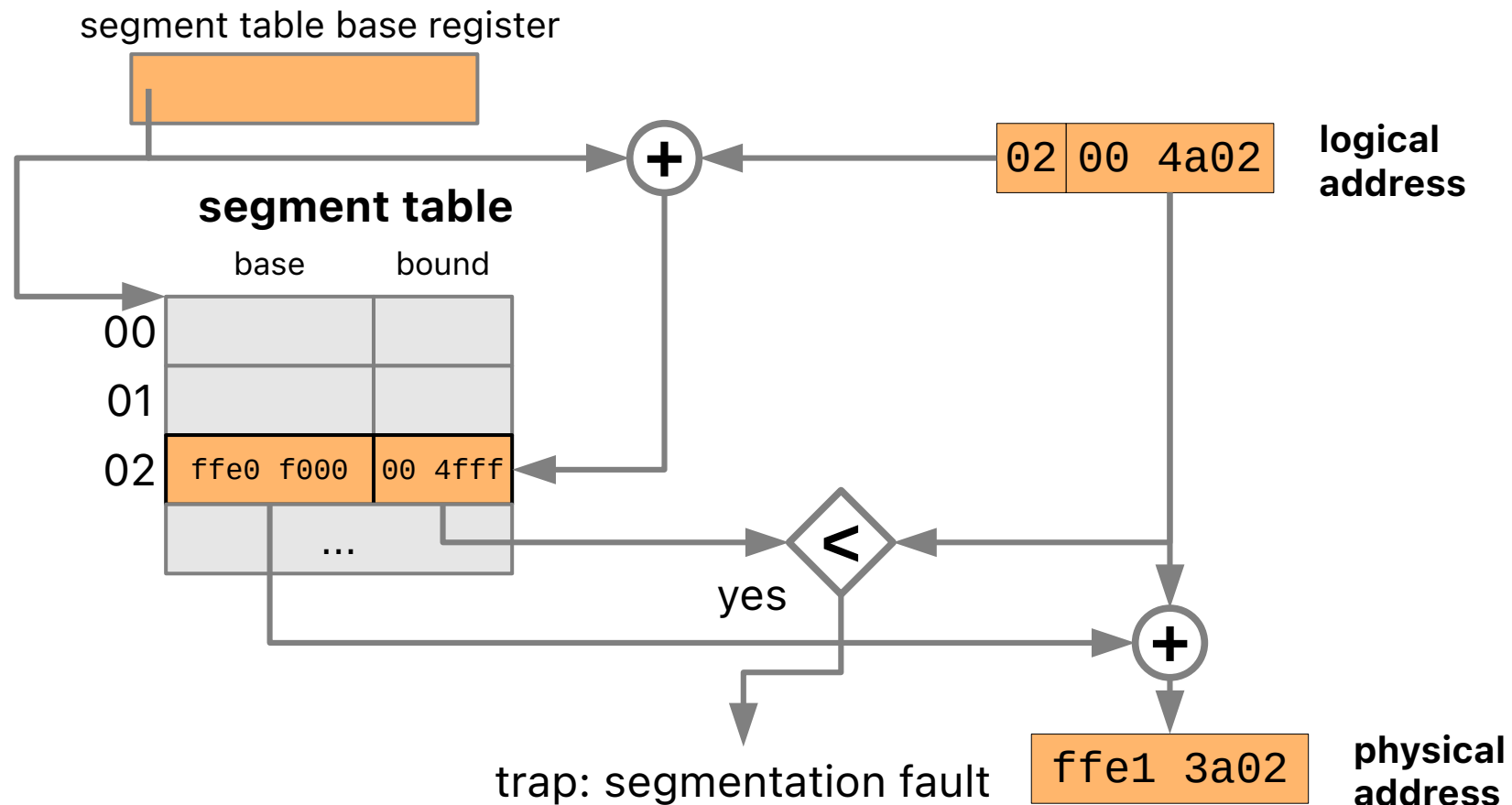
- realisation with translation table (per process)





# Memory Management – Segmentation

- realisation with translation table (per process)



# Memory Management – Segmentation

- hardware support is provided and implemented by the **memory management unit (MMU)**
- the MMU provides protection for segmentation violations
  - verification of access rights to **read**, **write** and **execute** commands depending on segments
  - **traps** indicate segmentation violations (→ segmentation fault)
  - programs and operating system are protected from each other
- replacing the segment base on each context switch
  - processes have their own translation table (→ stored in its PCB)
- simplification of swapping
  - after swapping-in only the segment table must be adjusted
- shared segments for text (program code) and data (shared memory) are possible

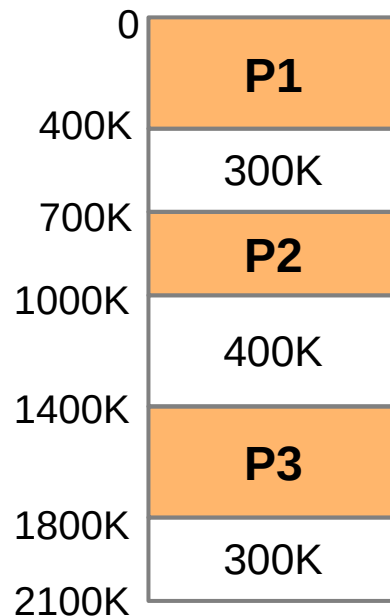
# Segmentation – Problems

- **memory fragmentation** due to frequent swapping-in/out (or start/termination) of processes
  - small, unusable holes occur (external fragmentation)
- **solution: compacting**
  - segments are moved to close holes
  - segment table needs to be adjusted and updated
  - performance penalty
- **issue: long I/O time overhead** due to swapping-in/out
  - not all parts of a segment are used equally often

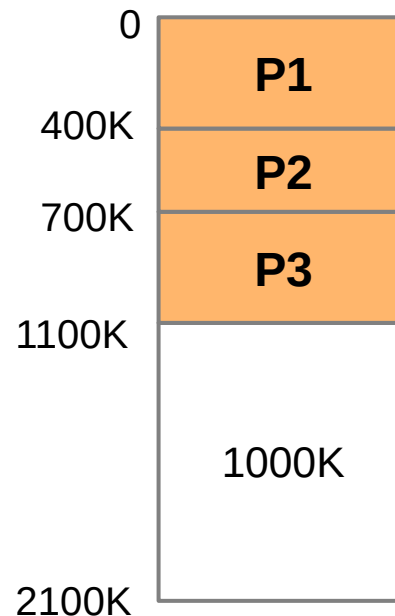
# Segmentation – Compacting

- move segments to defragment memory
  - creating fewer but larger gaps to reduce fragmentation
  - complex operation, depending on the size of the segments

original memory layout

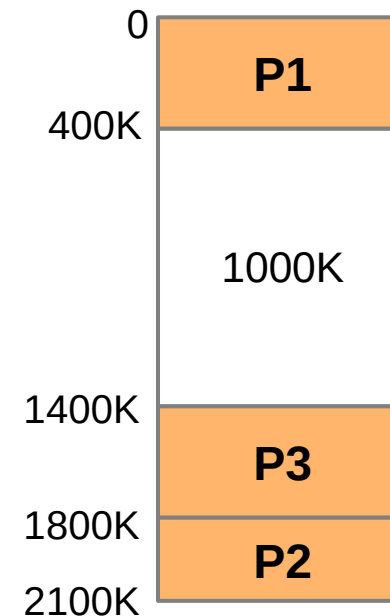


compacted layout #1



#1: move 700 K

compacted layout #2



#2: move 300 K

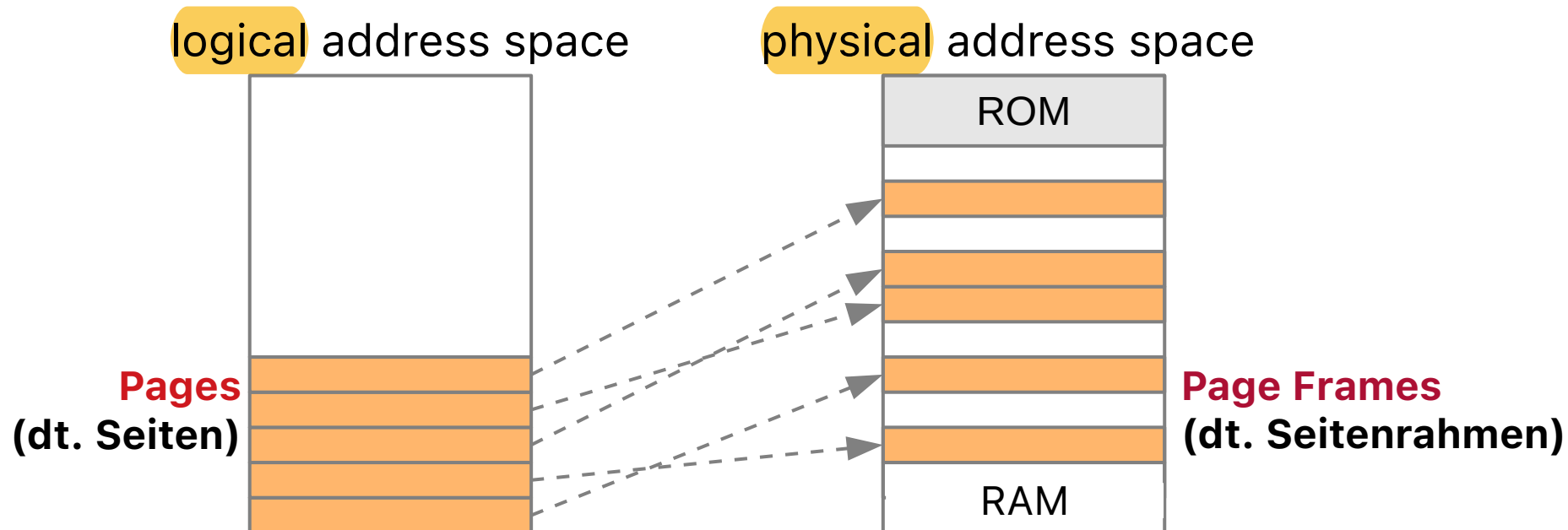
# Agenda

- ▶ Recap
- ▶ Organizational Matters
- ▶ Principles of Memory Management
  - ▶ Motivation
  - ▶ Requirements
  - ▶ Strategies
- ▶ Memory Allocation Schemes
- ▶ Memory Management for Multi-Program Operation
  - ▶ Segmentation
  - ▶ Paging
- ▶ Summary and Outlook



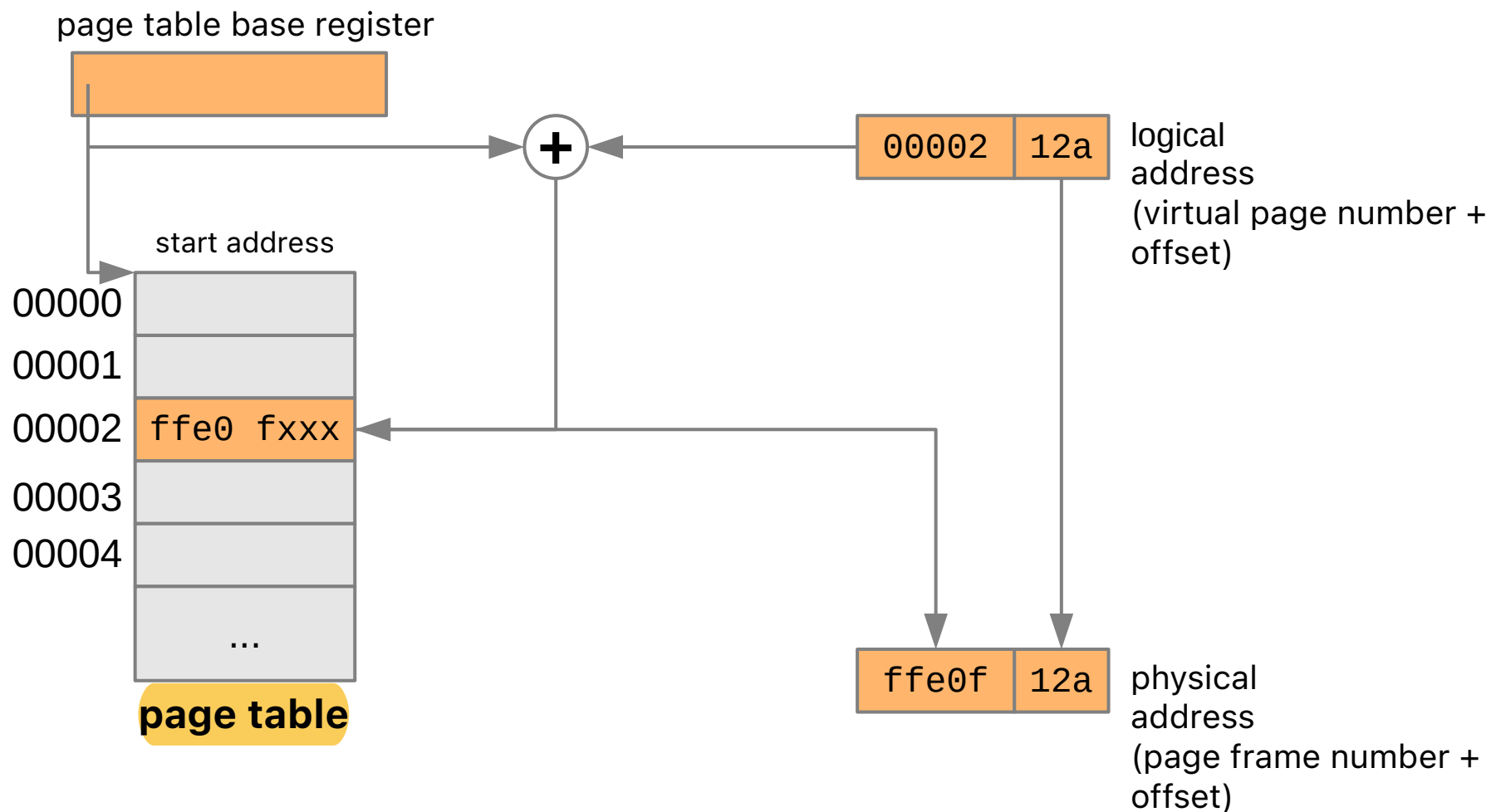
# Memory Management – Paging

- paging (dt. Seitenadressierung): organisation of the logical address space in pages of equal, **fixed size** (e.g., 4 K, 8 K)
- pages can be located at **any position** in the physical address space
  - solves the fragmentation problem
  - no more compacting necessary
  - **simplifies memory allocation and swapping in/out**



# Memory Management – Paging

- MMU translates logical (virtual page) to physical addresses (page frame)



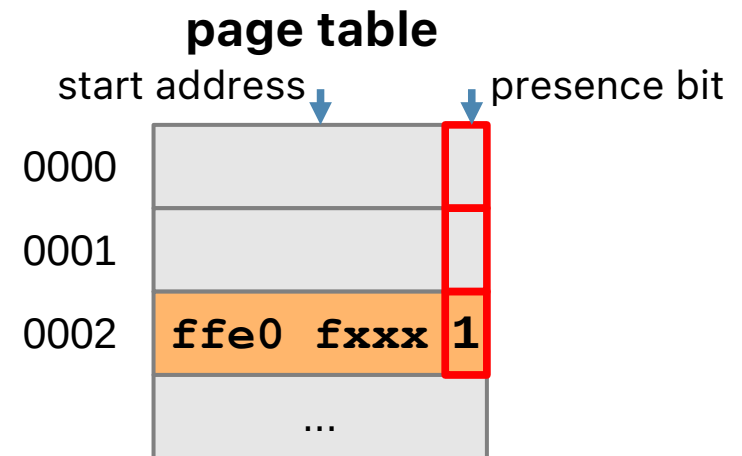
# Paging – Problems

- paging leads to **internal** fragmentation
  - last page may not be used completely
- page size
  - small pages reduce internal fragmentation, but increase size of page table (and vice versa)
  - common sizes: 512 bytes to 8192 bytes
- large table that must be kept in memory
- many implicit memory accesses necessary
  
- advanced paging: segmented paging, paged segmentation, inverted page tables



# Paging – Swapping In/Out of Pages

- it is no longer necessary to swap-in/out an entire segment
  - pages can be swapped in and out individually
- page faults, hardware support
  - if the **presence bit** is set, everything remains as before
  - if the presence bit is cleared, a trap is triggered (**page fault**)
  - trap handling can now be used for loading the page from the background memory and repeat the memory access afterwards (requires hardware support in the CPU)

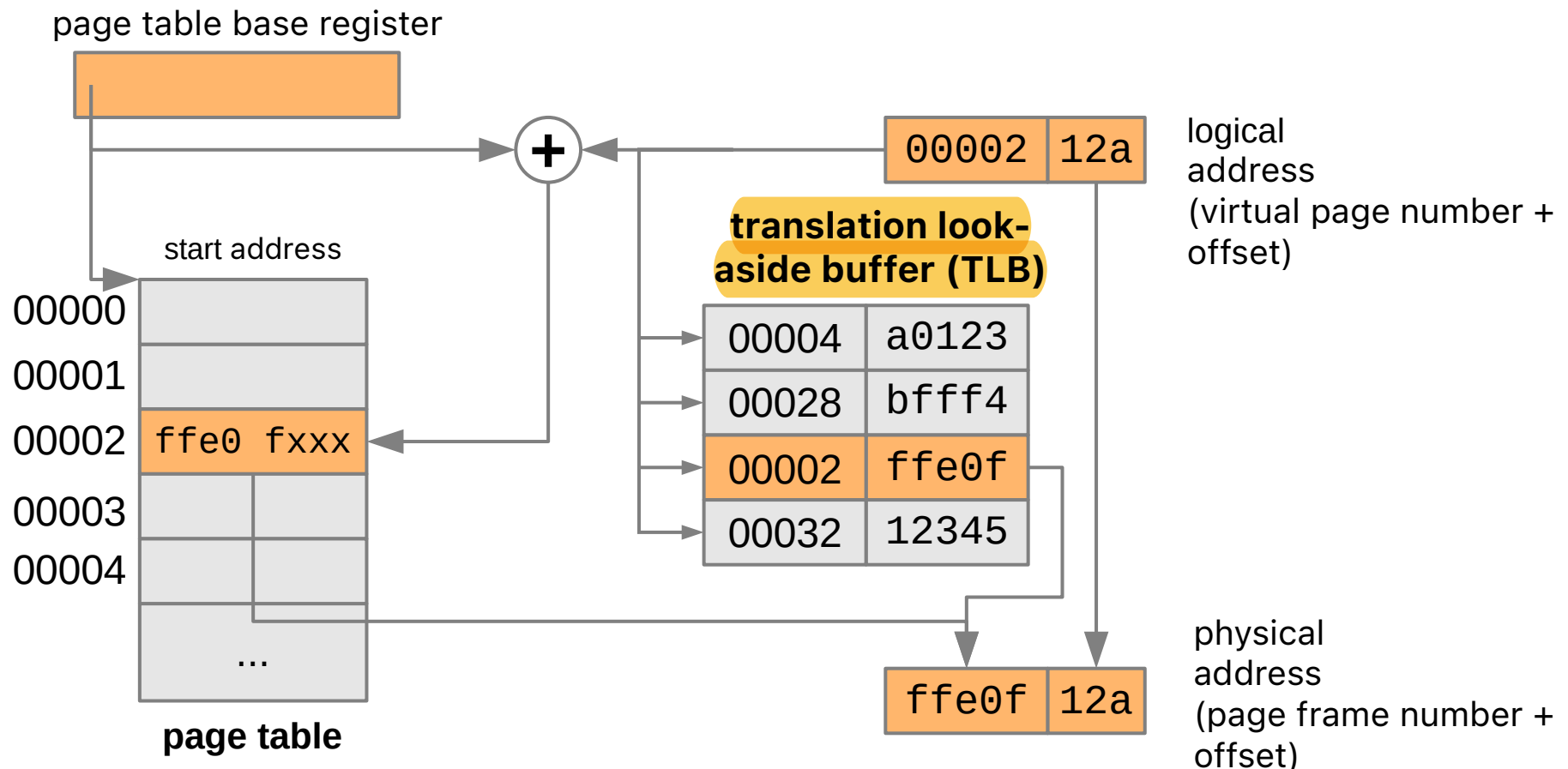


# Paging – Discussion on Operational Costs

- page faults may occur very frequently
  - depends on program and its I/O activity pattern(s)
  - Linux: `ps -eo min_flt,maj_flt,cmd`
- page translations (virtual to physical addresses) are expensive
  - example: memory reference
  - `movl 0x14711222, %eax`
    1. lookup page table base register (PTBR)
    2. extract virtual page number (VPN), here: `0x14711`
    3. read page table entry, here: `PTBR + 0x14711 * 8`
    4. extract page frame number (PFN), here: `0x222`
    5. read the memory at location `PFN << 12 + 0x222`
- **translation lookaside buffer (TLB): hardware cache, stores recent translations of virtual memory to physical memory addresses**

# Paging – Translation Lookaside Buffer (TLB)

- fast register set is consulted before accessing the page table:



# Paging – Translation Lookaside Buffer (TLB)

- fast access to pages if information is in TLB's memory
  - fully associative cache
  - TLB hit: no implicit memory accesses necessary
- TLB content must be replaced for context changes (TLB flush)
- when accessing a page not included in the TLB, the corresponding access information is entered into the TLB
  - an old entry must be selected for replacement
- typical TLB sizes
  - Intel Core i7: 512 entries, page size 4K
  - UltraSPARC T2: data TLB = 128, code TLB = 64, page size 8K
  - larger TLBs not possible at the usual clock rates at present

# Agenda

- ▶ Recap
- ▶ Organizational Matters
- ▶ Principles of Memory Management
  - ▶ Motivation
  - ▶ Requirements
  - ▶ Strategies
- ▶ Memory Allocation Schemes
- ▶ Memory Management for Multi-Program Operation
  - ▶ Segmentation
  - ▶ Paging
- ▶ Summary and Outlook



# ▶▶ Summary and Outlook

## ■ summary

- operating systems work closely with the hardware in terms of memory management
  - **segmentation** and/or **paging**
  - due to implicit indirection in memory access, programs and data can be moved arbitrarily during runtime under the control of the OS
- in addition, various strategic decisions are made
  - **placement strategy** (first fit, best fit, buddy, ...)
  - differences with regard to fragmentation as well as allocation and relocation costs
  - strategy selection depends on the usage pattern

## ■ outlook: virtual memory

- methods and strategies how/when segments or pages should be swapped in/out
- fetch strategies, replacement strategies

# References and Acknowledgments

## Lecture

- ▶ Systemnahe Programmierung in C (SPiC), Betriebssysteme (Jürgen Kleinöder, Wolfgang Schröder-Preikschat)
- ▶ Betriebssysteme und Rechnernetze (Olaf Spinczyk, Embedded Software Systems Group, Universität Osnabrück)

## Teaching Books and Reference Book

- [1] Avi Silberschatz, Peter Baer Galvin, Greg Gagne: *Operating System Concepts*, John Wiley & Sons, 2018.
- [2] Andrew Tanenbaum, Herbert Bos: *Modern Operating Systems*, Pearson, 2015.
- [3] Wolfgang Schröder-Preikschat: *Grundlage von Betriebssystemen – Sachwortverzeichnis*, 2023.  
<https://www4.cs.fau.de/~wosch/glossar.pdf>