System Architecture

16 Memory Management

RAM, Design Space, Examples

January 12 2009
Winter Term 2008/09
Gerd Liefländer

Recommended Reading

- Bacon, J.: Operating Systems (5)
- Bovet, D.: Understanding the Linux Kernel
- Knuth, D.: The Art of Computer Programming, Vol. 1, Ch. "Dynamic Storage Allocation"
- Nehmer, J.: Grundlagen modener BS, (4)
- Silberschatz, A.: Operating System Concepts (7)
- Stallings, W.: Operating Systems (7)
- Tanenbaum, A.: Modern Operating Systems (4)

Agenda

- Motivation
- Architecture of RAM Management
- Design Parameters
- Example Memory Managers
  - Ring Buffer
  - Stack
  - Boundary Tag Systems
  - Buddy System
  - Slab Allocating
  - Heap Management

Why Memory Management?

1. Each application needs RAM to run on a CPU, ⇒ we have to establish
   - static address regions, e.g. code, global data
   - dynamic address regions, e.g. heap or stack
2. The kernel needs memory for its
   - residentpart
   - loadable kernel modules
3. Devices often only can use physically addressed memory for their buffers etc.

⇒ Every executable needs some RAM

Why Memory Management?

<u>Entities of an "application" AS (address regions):</u>

- Code ("text segment" in Unix jargon)
- Data ((un)initialized data segment)
- Heap
- Stack

<u>Data types (entities) of the kernel AS*:</u>

- Buffer
- TCB, page table, free list, bit map, …

*At the end of the course you should be able to enumerate at least 10 different kernel data types

System Goals concerning Memory

- Increase (maximize)
  - memory utilization
  - Sometimes we must reserve some RAM capacity for
    - high priority applications or
    - system emergency functions
- Reduce (minimize)
  - application's response time
  - application's turnaround time
  - memory manager's overhead

What does a Memory Manager?

- Keeping track of memory that is currently
  - allocated, i.e. in use or reserved for future use
    - Pinning parts of memory for specific tasks
  - free
- Looking for fitting free memory in case of a request
  - If found, allocate free memory according to some policy
- Free memory in case of a release
  - potentially look for free neighbors in order to reunify free neighbored memory pieces

Memory Management (1)

- Programmers want memory being
  - large & fast & non volatile
- *Current technology* does not support all of these at once, but *future* technologies like MRAM* might do
- System architects offer a *memory hierarchy*
  - small high-speed caches (expensive)
  - medium sized fast main memory (RAM)
  - flash memory
  - Giga bytes of slow, cheap disk storage
  - Terra bytes of very slow archive memory

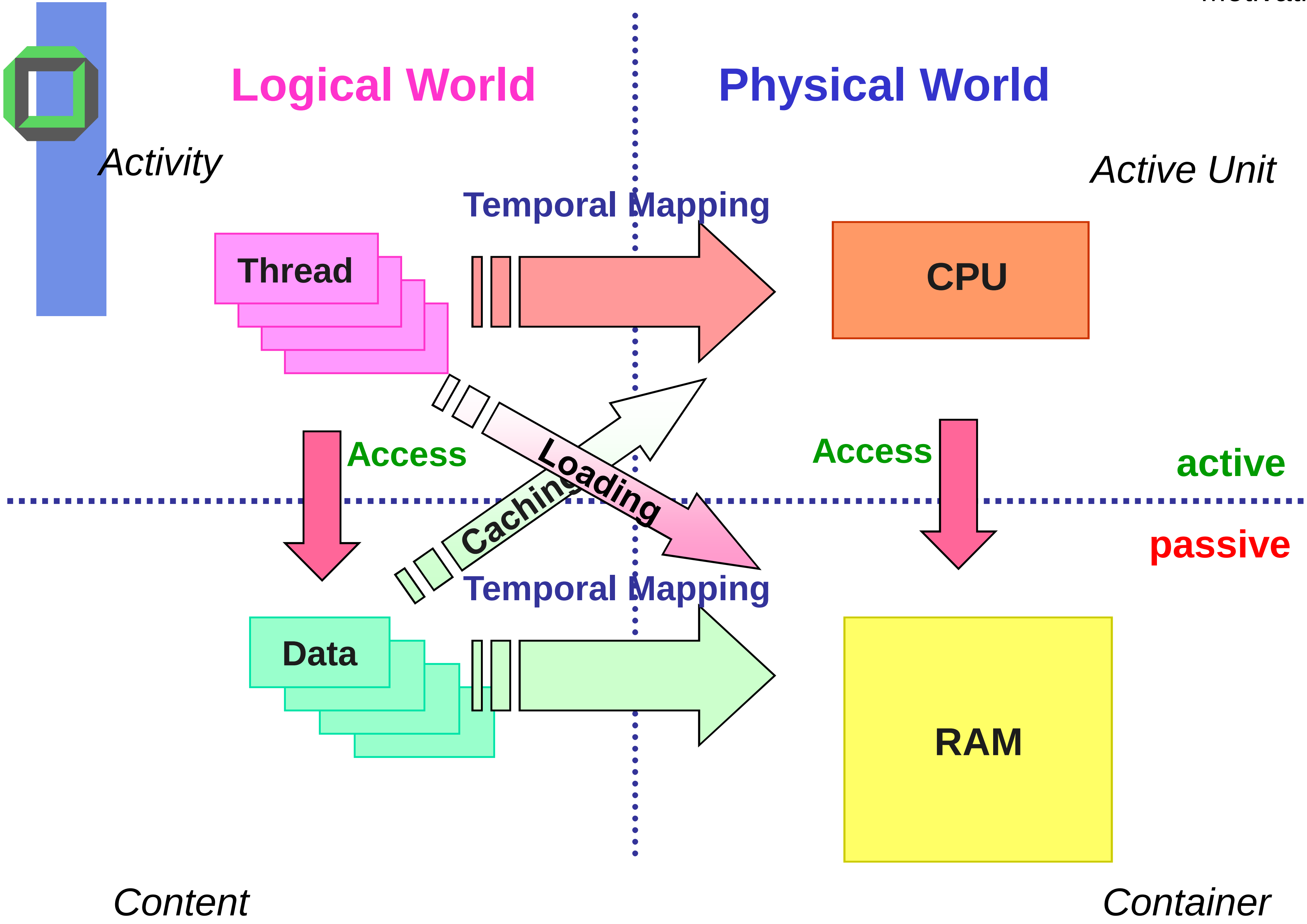
*MRAM = Magnetic RAM

Memory Management (2)

Two main goals:

1. Manage memory efficiently
   - appropriate algorithms & data structures
   - program MMU (e.g. TLB)

2. Establish effective usability of memory
   - maximize usage of (main) memory
   - support low cache footprint


Logical World
Physical World
Activity
Active Unit
Temporal Mapping
Thread
CPU
Access
Loading
Caching
Access
active
passive
Temporal Mapping
Data
RAM
Content
Container

Why to bother about RAM?

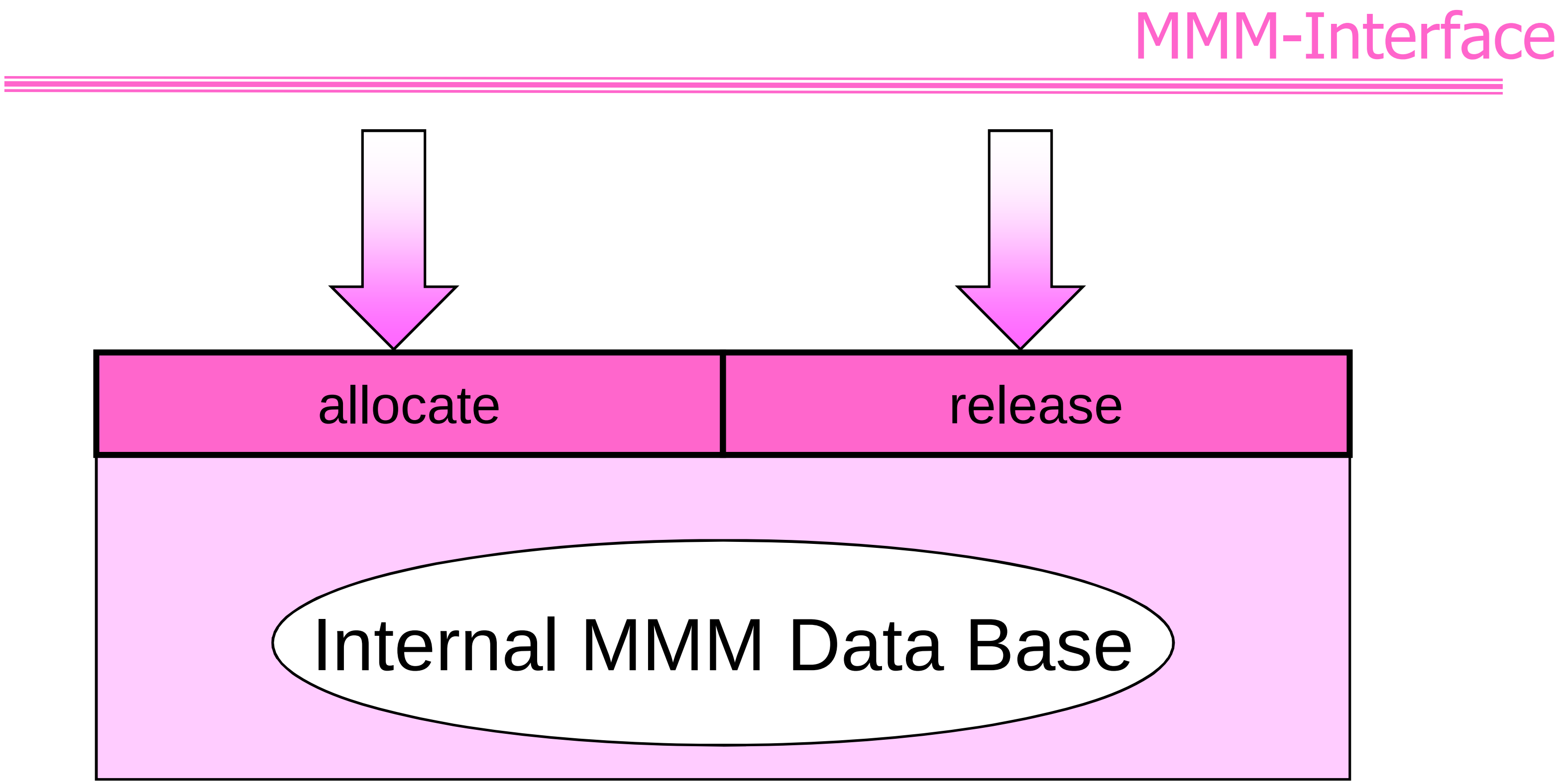
Memory is large <u>and</u> cheap, and if not, we'll use virtual memory.

<u>However, reality tells us:</u>

1. Many computers don't use virtual memory at all
2. Modern programs tend to be memory greedy & virtual memory ≠ unlimited memory
3. Memory management has to be done anyway at
   - system level
   - application level

Architecture of RAM Management

Memory Management Module

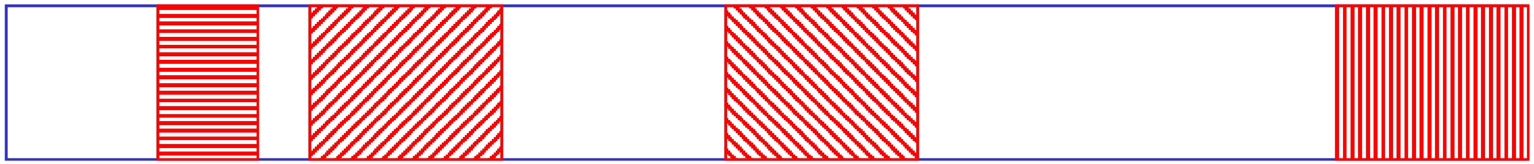


Note:

∃ several related and/or unrelated memory managers

MMM Data Base

MMMDB reflects what parts are allocated and what parts are free

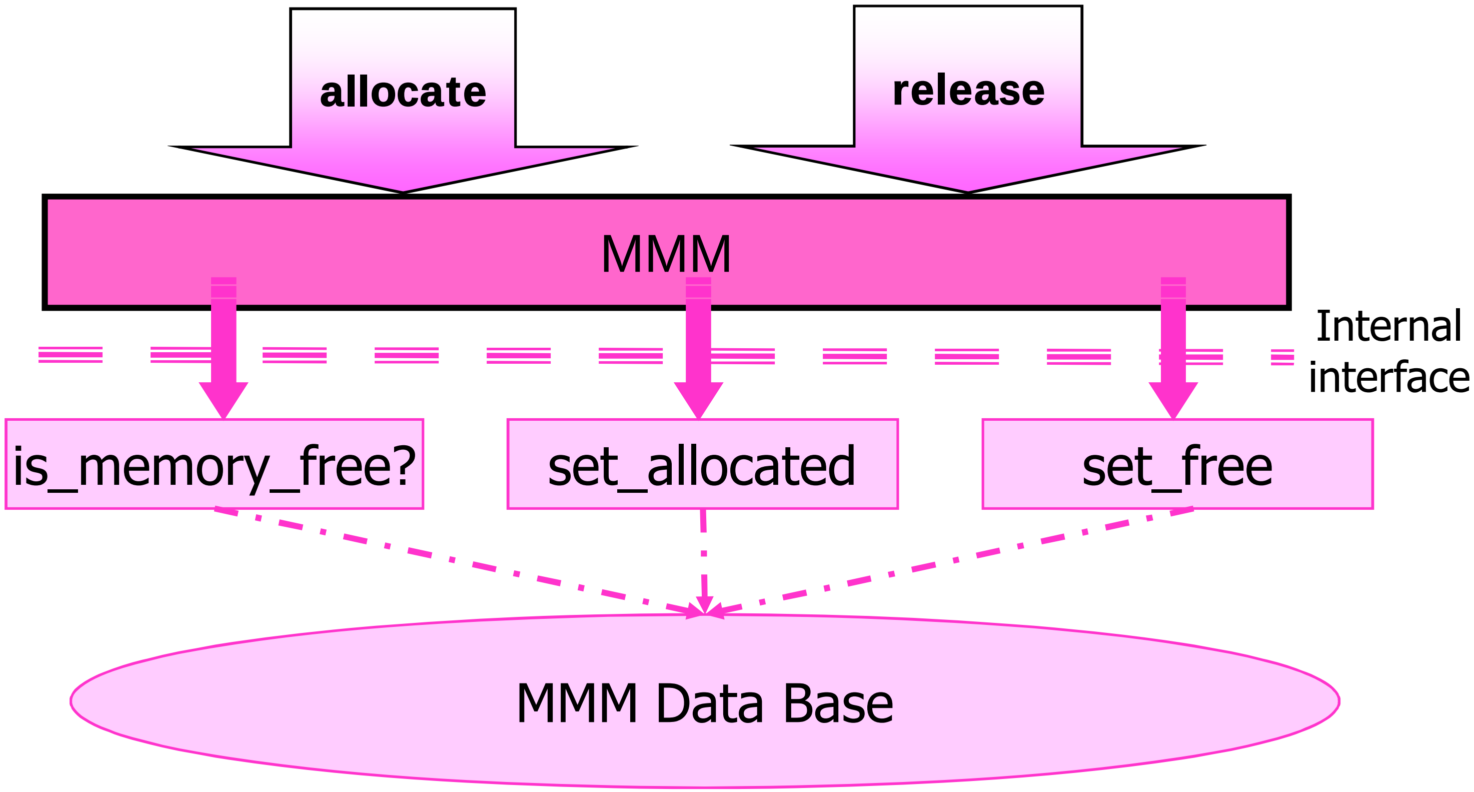


We must solve the following problems when handling memory requests

- Efficiently select a fitting free memory block
  - As fast as possible
- Try to utilize the memory efficiently
  - Avoid unusable memory leaks
- Meet additional constraints
  - Avoid unbounded waiting in front of MMM

Smart System Architecture of a MMM

MMM-Interface



Main advantage of the above architecture?

Design Parameters

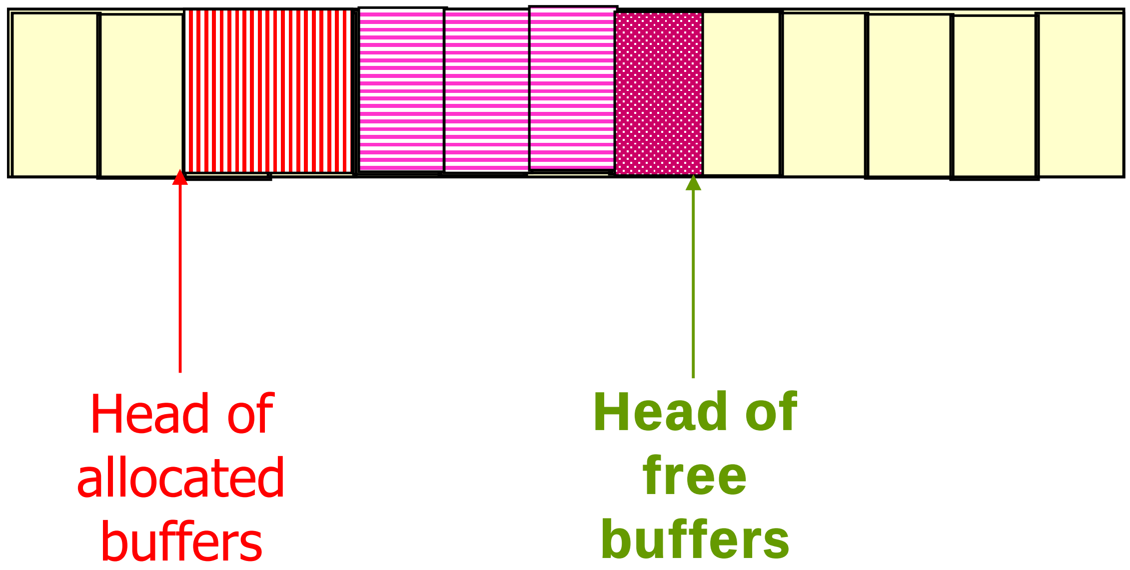
Orthogonal Parameters

- Sequence of allocate/release-operations
- Size of memory blocks
- Data structures
- *Fragmentation (not design parameter but result of a design!!!!)*
- Allocation policy
- Reunification of released blocks
- ...?

Playground for an innovative system architect

MM Design Parameters (1a)

- Sequence of allocate/release-operations
  - ~FIFO = queue



MM Design Parameters (1b)

- Sequence of allocate/release-operations
  - FIFO, LIFO = stack

MM Design Parameters (1c)

- Sequence of allocate/release-operations
  - FIFO, LIFO, arbitrary = in general

Additionally we have to solve the problem of memory holes

MM Design Parameters (2)

- Sequence of allocate/release-operations
  - FIFO, LIFO, arbitrary
- Size of memory blocks

MM Design Parameters (2a)

- Sequence of allocate/release-operations
  - FIFO, LIFO, arbitrary
- Size of memory blocks
  - Constant size = most buffering

MM Design Parameters (2b)

- Sequence of allocate/release-operations
  - FIFO, LIFO, arbitrary
- Size of memory blocks
  - Constant size, multiple of fixed size

MM Design Parameters (2c)

- Sequence of allocate/release-operations
  - FIFO, LIFO, arbitrary
- Size of memory blocks
  - Constant size, multiple size, fixed size = reservoir of frequently used blocks

MM Design Parameters (2d)

- Sequence of allocate/release-operations
  - FIFO, LIFO, arbitrary
- Size of memory blocks
  - Identical size, multiple size, fixed size, exponential size = buddy system

MM Design Parameters (2e)

- Sequence of allocate/release-operations
  - FIFO, LIFO, arbitrary
- Size of Memory blocks
  - Identical size, multiple size fixed size, exponential size, arbitrary

MM Design Parameters (3a)

- Sequence of allocate/release-operations
  - FIFO, LIFO, arbitrary
- Size of memory blocks
  - Identical size, multiple size, fixed size, exponential size, arbitrary
- Data structures
  - Integrated within memory block(s) = mostly with larger blocks

MM Design Parameters (3b)

- Sequence of allocate/release-operations
  - FIFO, LIFO, arbitrary
- Size of memory blocks
  - Identical size, multiple size, fixed size, exponential size, arbitrary
- Data structures
  - Integrated, special memory block(s) = stack

Result of Design (4)

- Sequence of allocate/release-operations
  - FIFO, LIFO, arbitrary
- Size of memory blocks
  - Identical size, multple size, fixed size, exponential size, arbitrary
- Data Structures
  - Integrated, special memory block(s)
- Fragmentation
  - With(out) in(ex)ternal fragmentation

MM Design Parameters (5a)

- Sequence of allocate/release-operations
  - FIFO, LIFO, arbitrary
- Size of memory blocks
  - Identical size, multiple size, fixed size, exponential size, arbitrary
- Data structures
  - Integrated, special memory block(s)
- Fragmentation
  - With(out) in(ex)ternal fragmentation
- Allocation policy
  - First-Fit = take the **first fitting block** within an **ordered set of free blocks**

How would you order the set of free blocks?

MM Design Parameters (5b)

- Sequence of allocate/release-operations
  - FIFO, LIFO, arbitrary
- Size of memory blocks
  - Identical size, multiple size, fixed size, exponential sized, arbitrary
- Data structures
  - Integrated, special memory block(s)
- Fragmentation
  - With(out) in(ex)ternal fragmentation
- Allocation policy
  - First-, Next-Fit = take the next fitting block within an ordered set of free blocks

MM Design Parameters (5c)

- Sequence of allocate/release-operations
  - FIFO, LIFO, arbitrary
- Size of memory blocks
  - Identical size, multiple size, fixed size, exponential sized, arbitrary
- Data structures
  - Integrated, special memory block(s)
- Fragmentation
  - With(out) in(ex)ternal fragmentation
- Allocation policy
  - First-, Next-, BestFit = take the best fitting block within an ordered set of free blocks

MM Design Parameters (5d)

- Sequence of allocate/release-operations
  - FIFO, LIFO, arbitrary
- Size of memory blocks
  - Identical size, multiple size, fixed size, exponential sized, arbitrary
- Data structures
  - Integrated, special memory block(s)
- Fragmentation
  - With(out) in(ex)ternal fragmentation
- Allocation policy
  - First-, Next-, Best-, Worst-Fit = take the largest fitting block within an ordered set of free blocks

MM Design Parameters (5e)

- Sequence of allocate/release-operations
  - FIFO, LIFO, arbitrary
- Size of memory blocks
  - Identical size, multiple size, fixed size, exponential sized, arbitrary
- Data structures
  - Integrated, special memory block(s)
- Fragmentation
  - With(out) in(ex)ternal fragmentation
- Allocation policy
  - First-, Next-, Best-, Worst-, Nearest-Fit, = take the fitting block closest to the previous fitting block

MM Design Parameters (6a)

- Sequence of allocate/release-operations
  - FIFO, LIFO, arbitrary
- Size of memory blocks
  - Identical size, multiple size, fixed size, exponential size, arbitrary
- Data structures
  - Integrated, special memory block(s)
- Fragmentation
  - With(out) in(ex)ternal fragmentation
- Allocation policy
  - First-, Next-, Best-, Worst-, Nearest-Fit
- **Reunification** of released blocks
  - **Eager reunification** with neighbored free blocks (if any)

MM Design Parameters (6b)

- Sequence of allocate/release-operations
  - FIFO, LIFO, arbitrary
- Size of memory blocks
  - Identical size, multiple size, fixed size, exponential size, arbitrary
- Data structures
  - Integrated, special memory block(s)
- Fragmentation
  - With(out) in(ex)ternal fragmentation
- Allocation policy
  - First-, Next-, Best-, Worst-, …, Nearest-Fit
- Reunification of Released blocks
  - Eager, **lazy reunification** = wait a while until you reunify

Summary of Designing MM

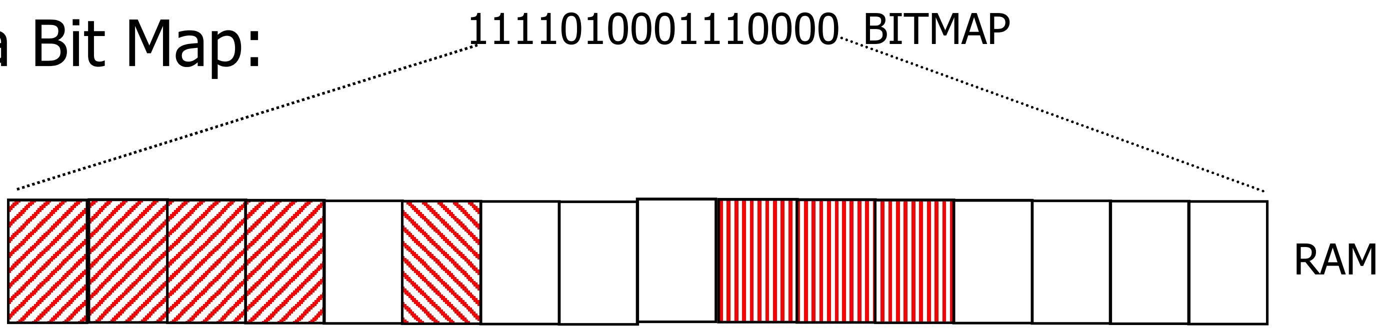
- Sequence of allocate/release-operations
  - FIFO, LIFO, arbitrary
- Size of memory blocks
  - Identical size, multiple size, fixed size, exponential size, arbitrary
- Data structures
  - Integrated, special memory block(s)
- Fragmentation
  - With(out) internal or external fragmentation
- Allocation policy
  - First-, Next-, Best-, Worst-, ..., Nearest-Fit
- Reunification of Released blocks
  - Immediate, lazy reunification

Data Structures

- Bit Map (for fixed sized units, e.g. pages)
  - Extra data structure
  - Integrated
- Table/List (for arbitrary sized units, e.g. segments)
  - Extra data structure
  - Integrated

Bit Map Data Base

Extra Bit Map:



Overhead per bit map:
The smaller the memory units, the larger the bit map

Integrated Bit Map:

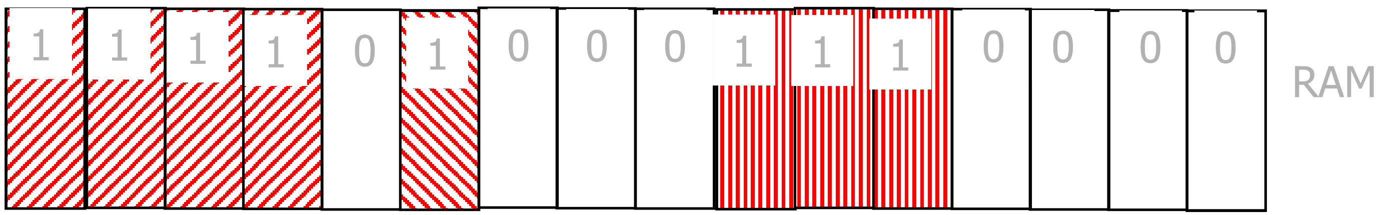


Table Data Base

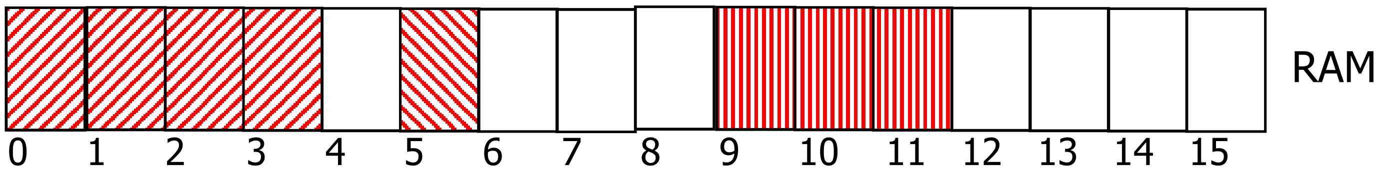
Extra Tables (sorted according to addresses):

Free Table:

Address	Length
4	1
6	3
12	4

Allocate Table:

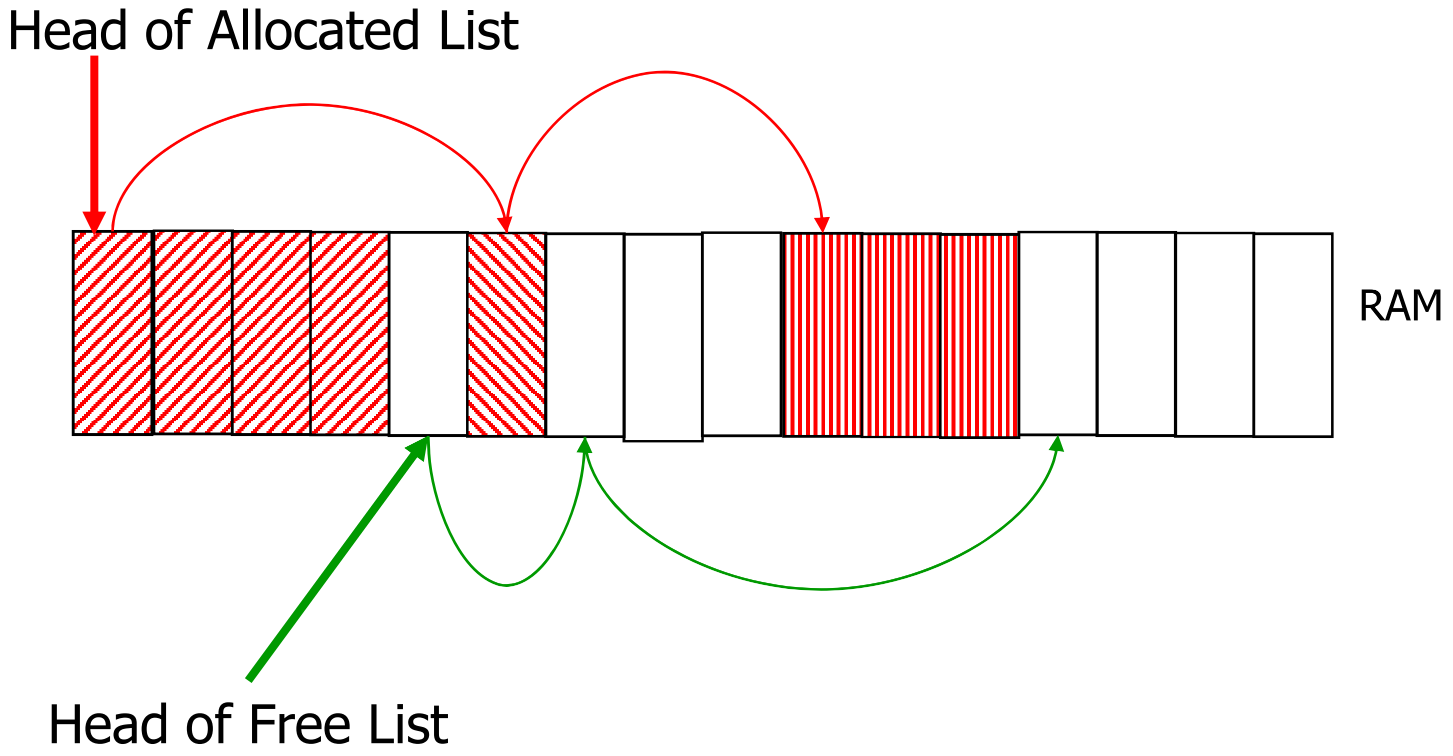
Address	Length
0	4
5	1
9	3



∃ other useful sorting criteria?

List oriented Data Base

Integrated List (sorted according to addresses):



Fragmentation

Internal Fragmentation:

waste

- allocated memory can be larger than requested memory
- memory management rounds up requested memory to the next manageable memory block unit

External Fragmentation:

loss

- Sum of total free memory space could satisfy a request, but free memory is scattered and is not contiguous

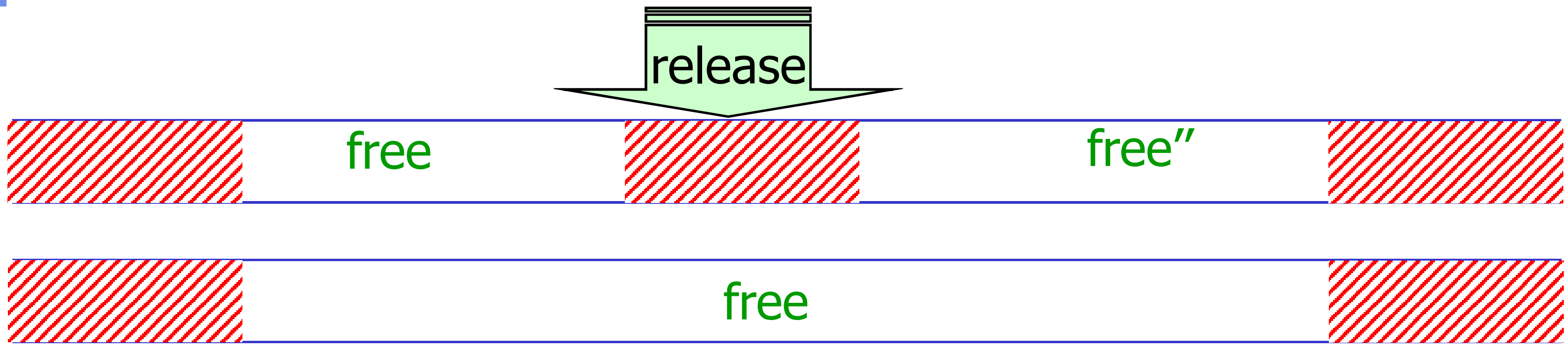
Allocation Policies

- First-Fit
- Next-Fit (Rotating First-Fit)
- Best-Fit
- Nearest-Fit

Analyze pros and cons of each of them

Reunification Policies

- Eager reunification (when releasing memory)



- Lazy reunification

Additional Requirements

- With lazy reunification MM neighboring blocks can be free. However, none of these can satisfy the current memory request:

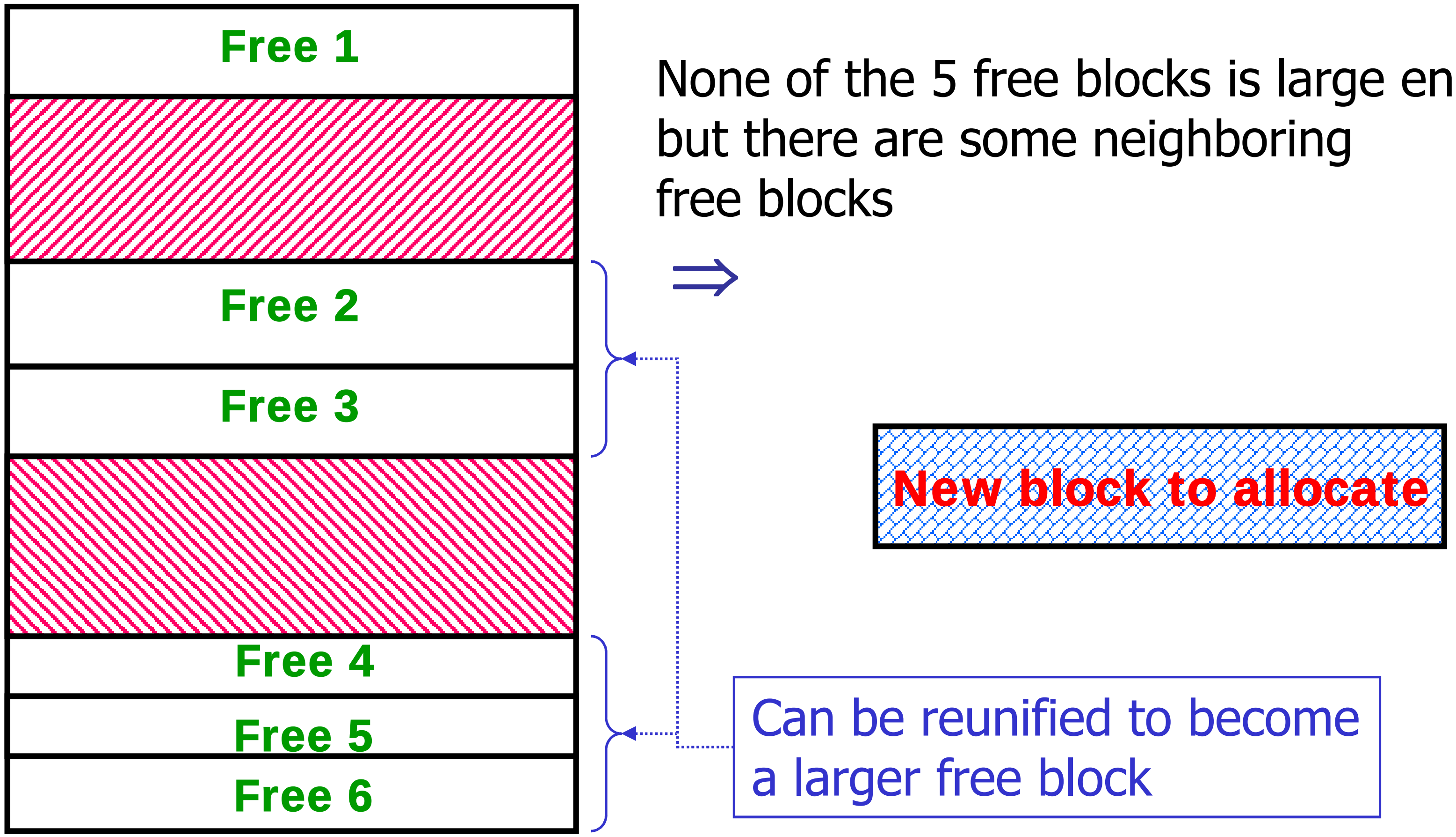
  ⇒ Garbage Collection[1]

- With an arbitrary allocation scheme (e.g. pure segmentation) we might get external fragmentation with a lot of scattered free blocks (Swiss cheese)

  ⇒ Compaction

[1]Garbage collection = releasing memory of no longer referenced objects

Garbage Collection (1)

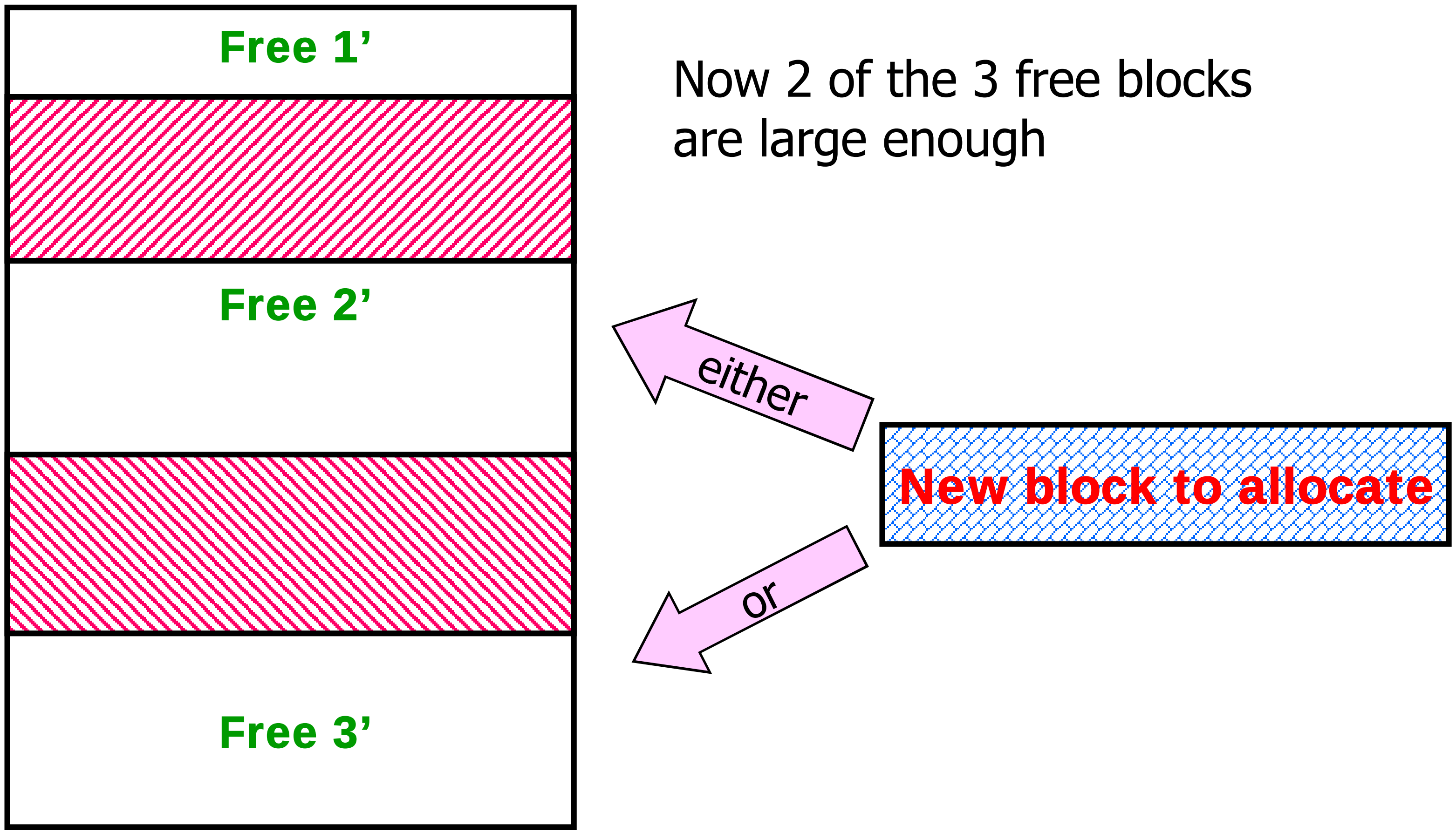


None of the 5 free blocks is large enough, but there are some neighboring free blocks

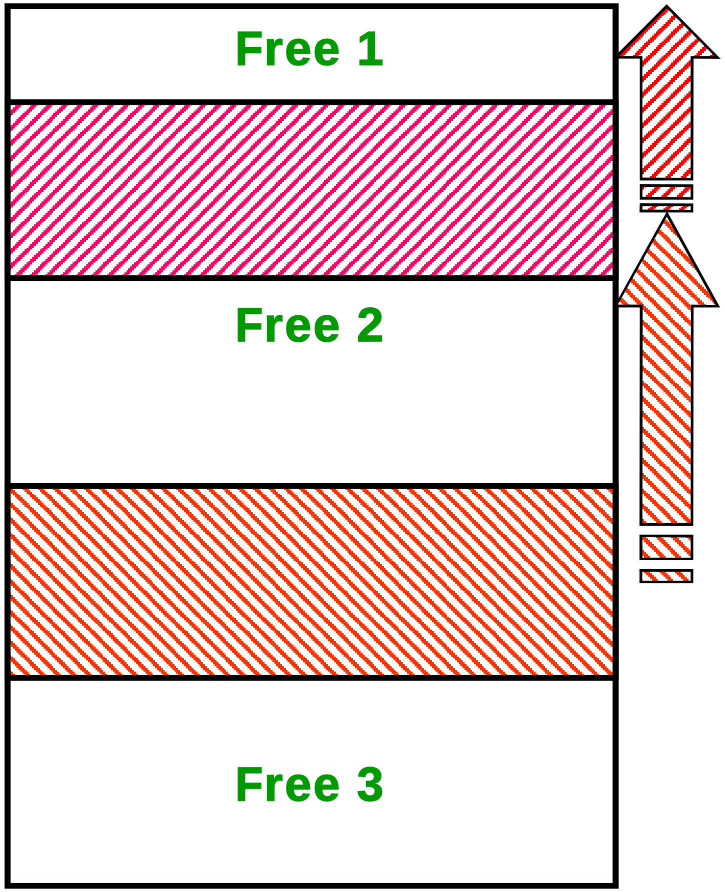
⇒

Garbage Collection (2)

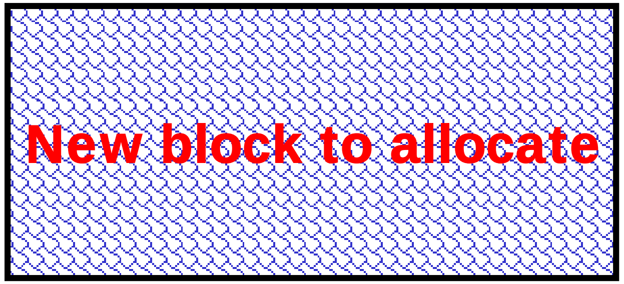
Now 2 of the 3 free blocks are large enough



Compaction (1)

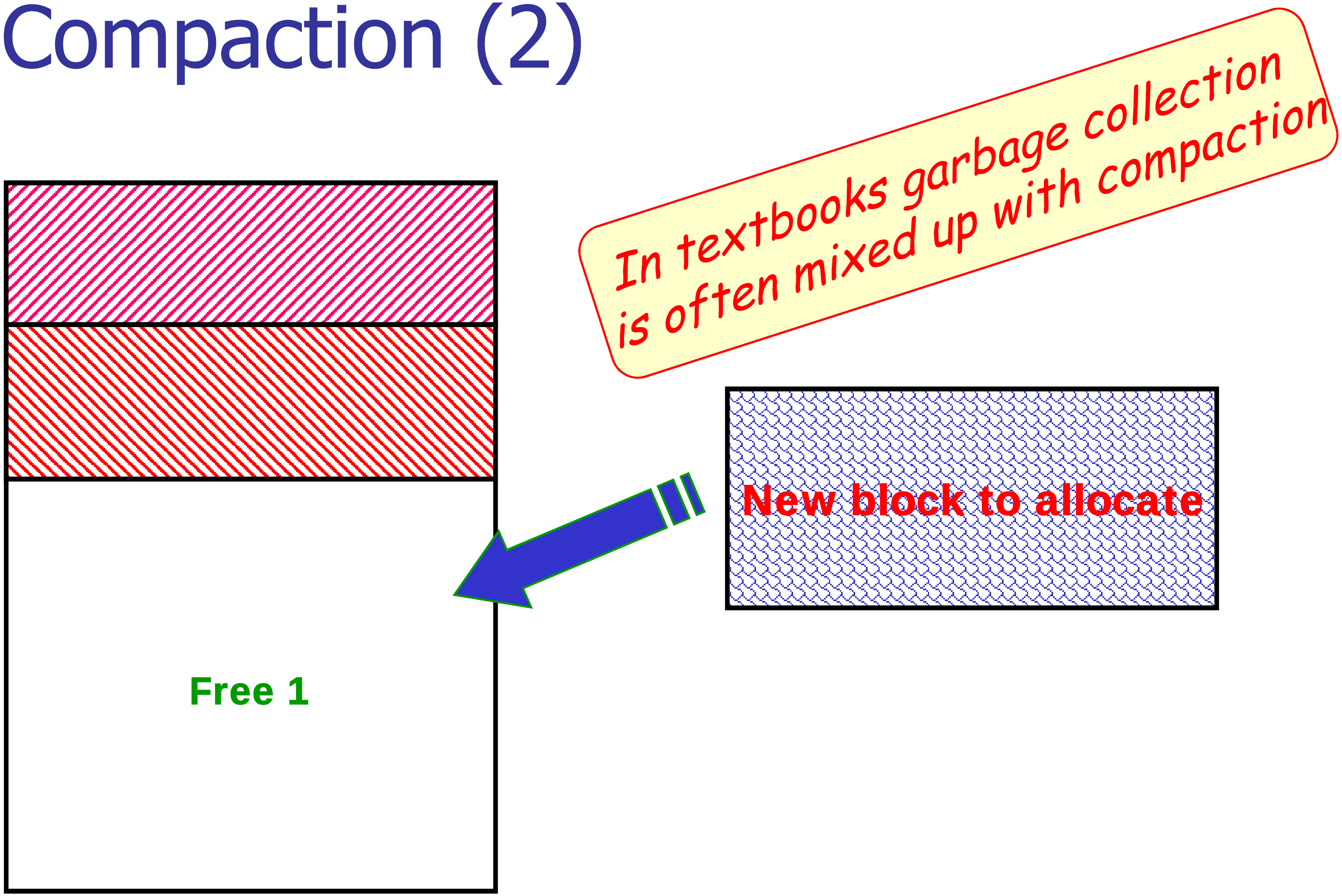


Observation:
None of the 3 free blocks is large enough, but ∃ enough free memory



Idea:
Move the allocated blocks towards a chosen memory boundary

Compaction (2)



Examples: Memory Manager

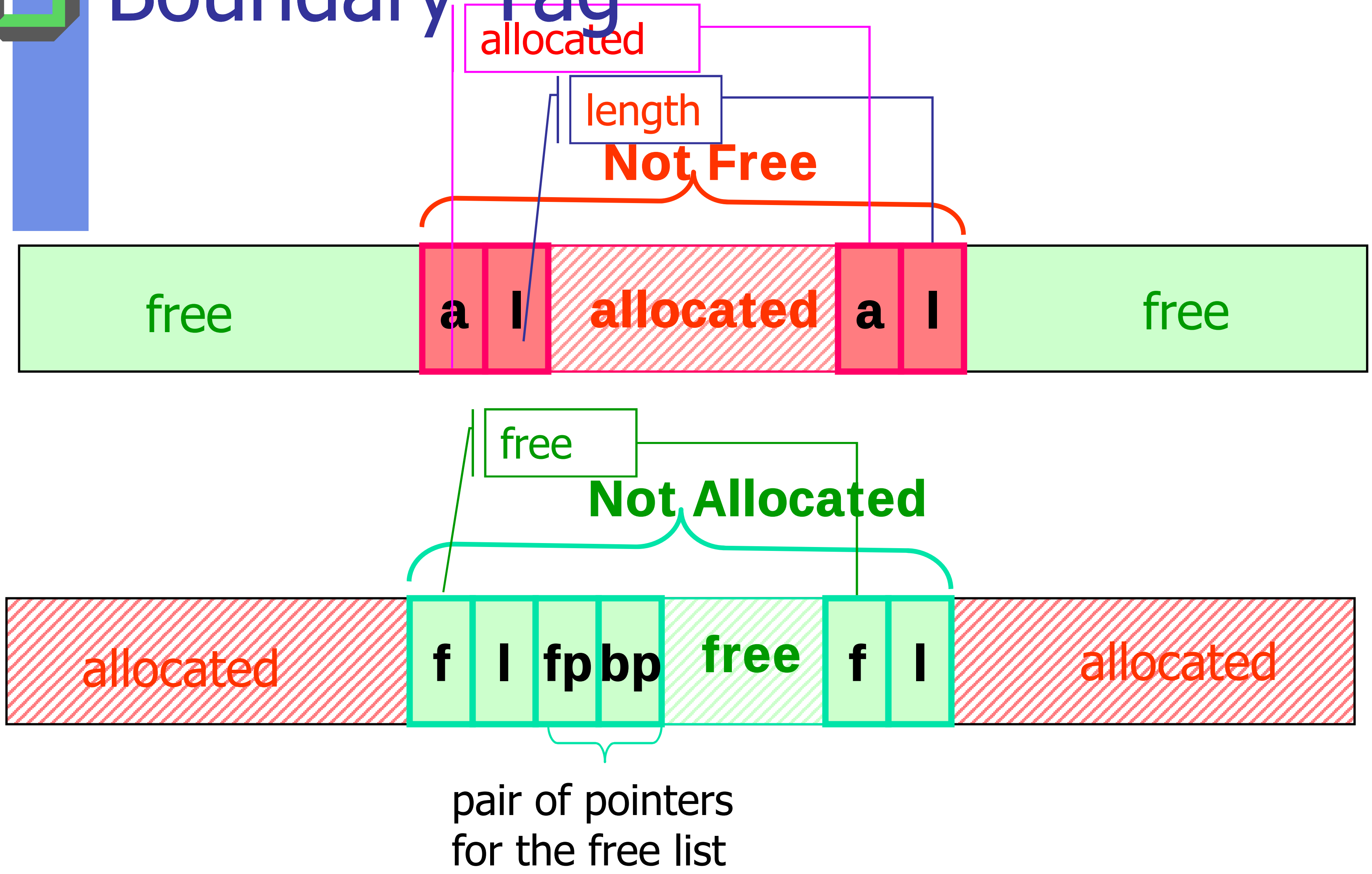
- Ringbuffer
- Stack

} Very specific design parameters

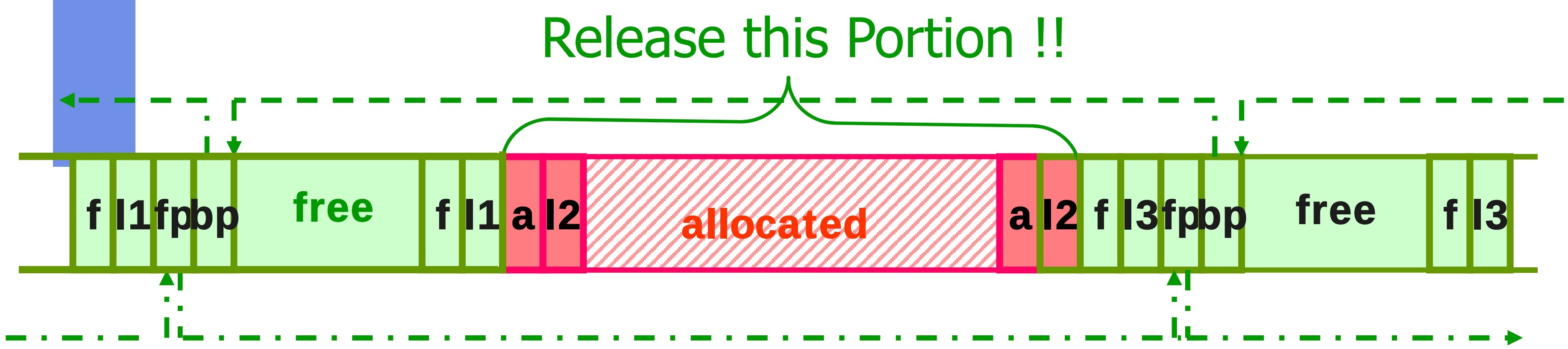
- Boundary Tag System (“Randkennzeichnungsverfahren”)
  - Operations in arbitrary order
  - Arbitrary sized blocks
  - Integrated management data structures
  - Allocation according to xyz-Fit (best-fit is possible)
  - External fragmentation
  - Immediate reunification
- Buddy System (“Halbierungsverfahren”)
  - Operations in arbitrary order
  - Allocated blocks of 2^0, 2^1, 2^2, 2^3, ...
  - Explicit management data structures
  - Allocation according to “Best-Fit”
  - Internal and external fragmentation
  - Immediate (or lazy) reunification
- Linux Slab Allocator (“Stückchenzuteiler”)

} Very specific design parameters

Boundary Tag



Reunification (1)

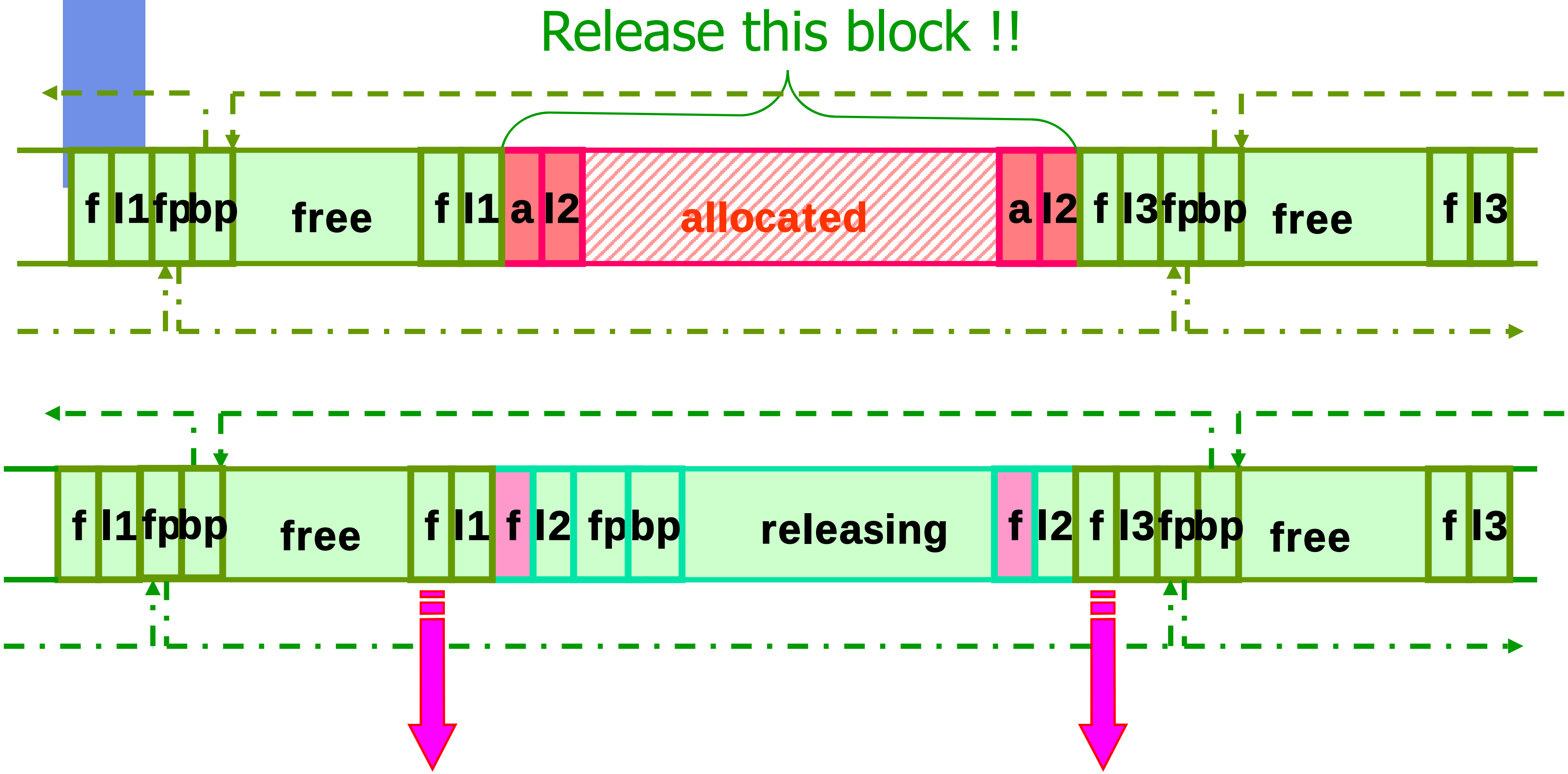


What to do?

Because length and status field are of fixed size ⇒ we can easily get the length of the block to be released.

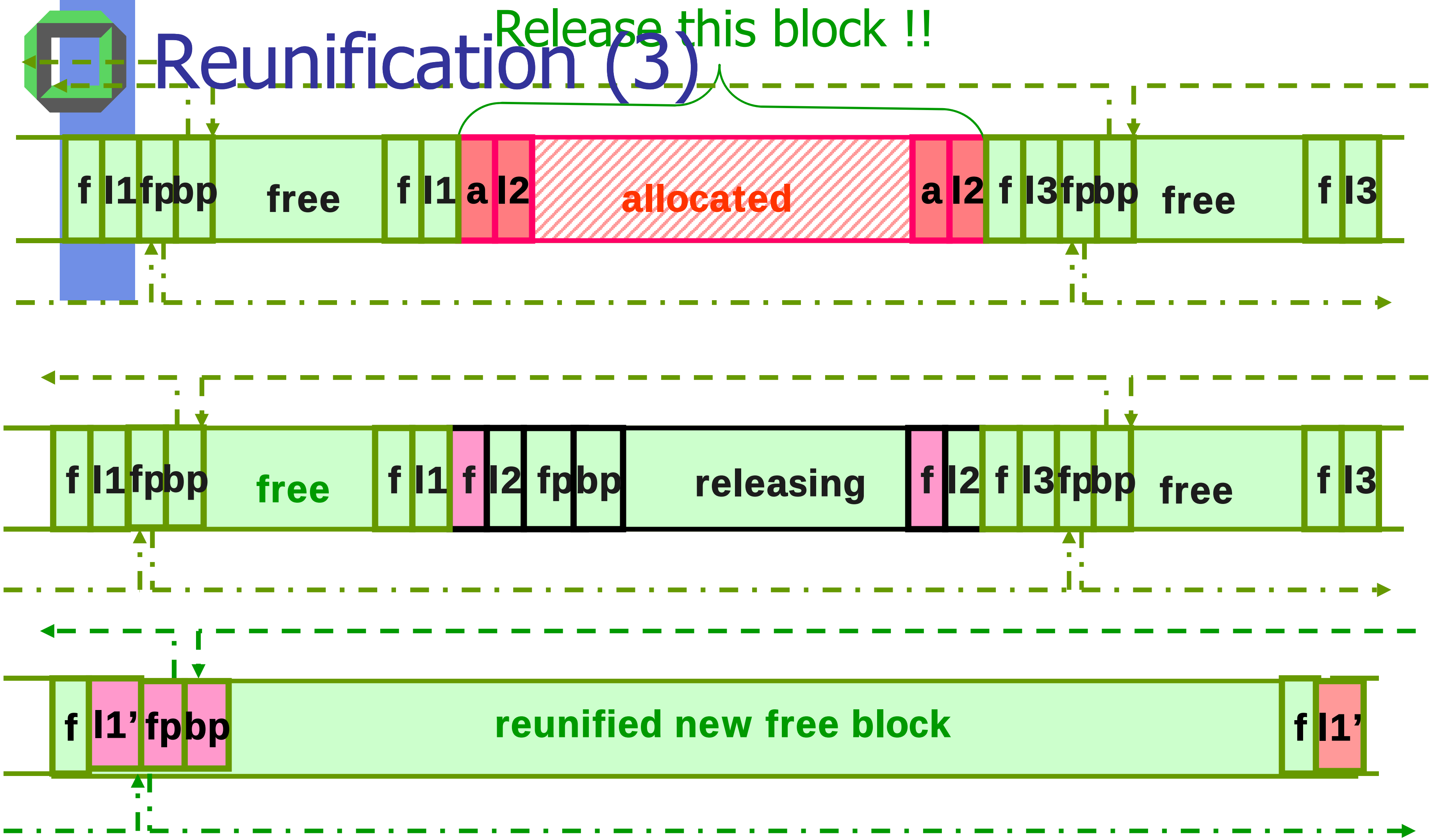
Furthermore we can look over both boundaries to get necessary information about the neighboring blocks.

Reunification (2)



We can detect whether both (one or no) neighbor(s) are(is) free ⇒

Reunification (3)



Finally we have to adjust the new pointers in the free list and the resulting length field (in this case: l1′ = l1 + l2 + l3)

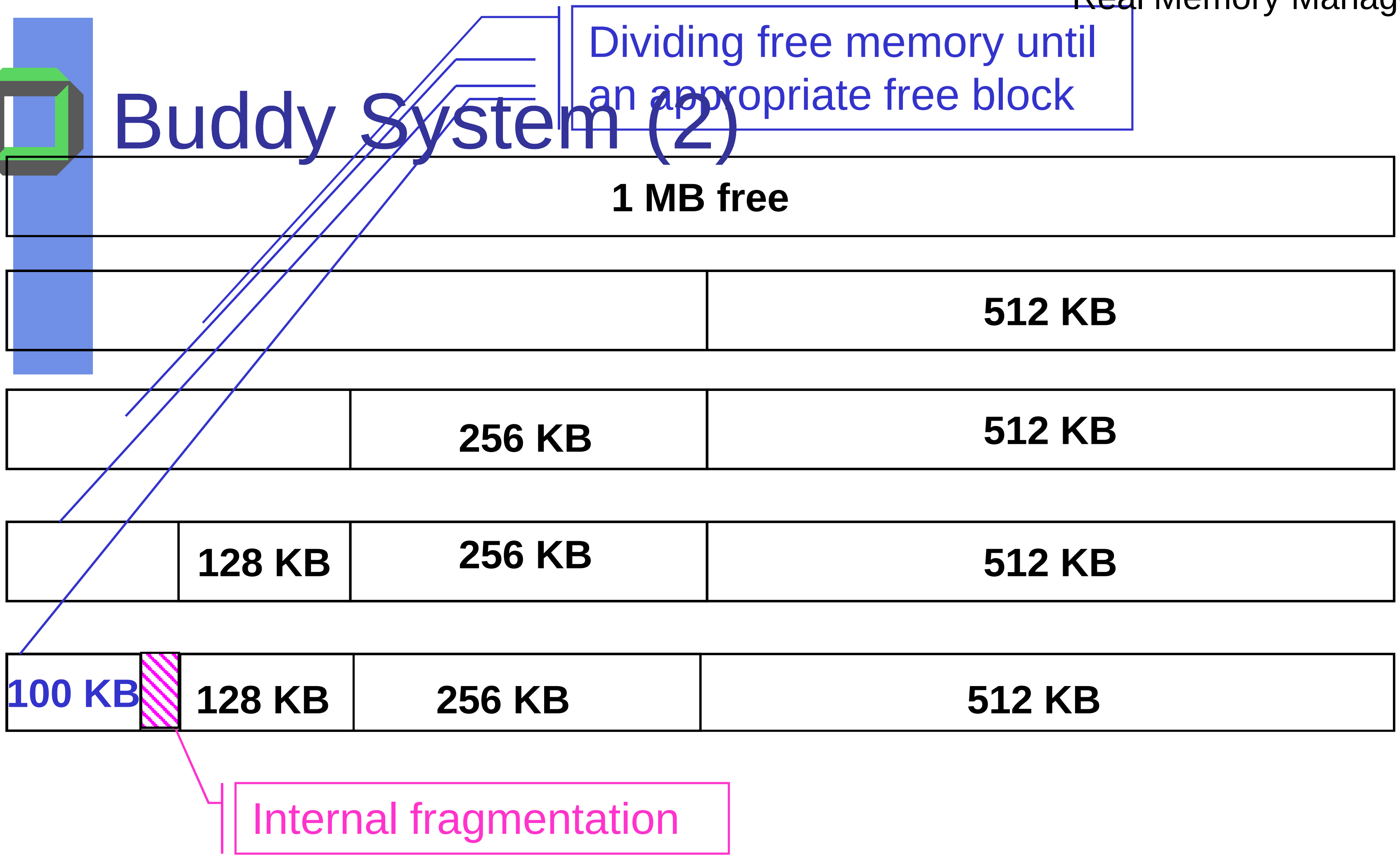
Buddy System (1)

Memory to be managed

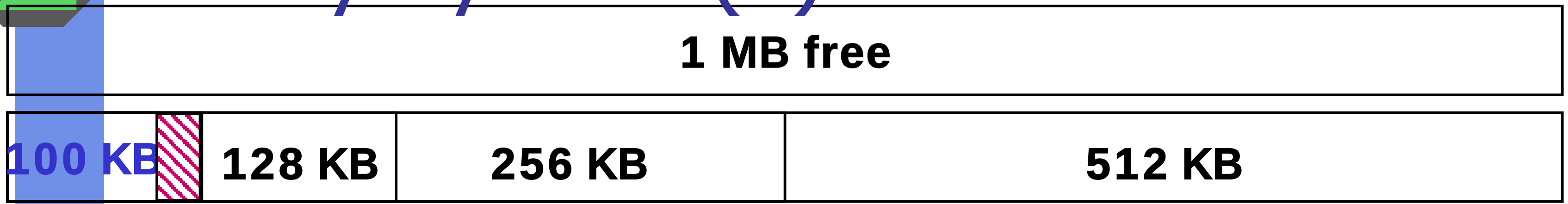
2^{20} **Byte = 1 MB free**

Suppose a client requests 100 KB

Buddy System (2)

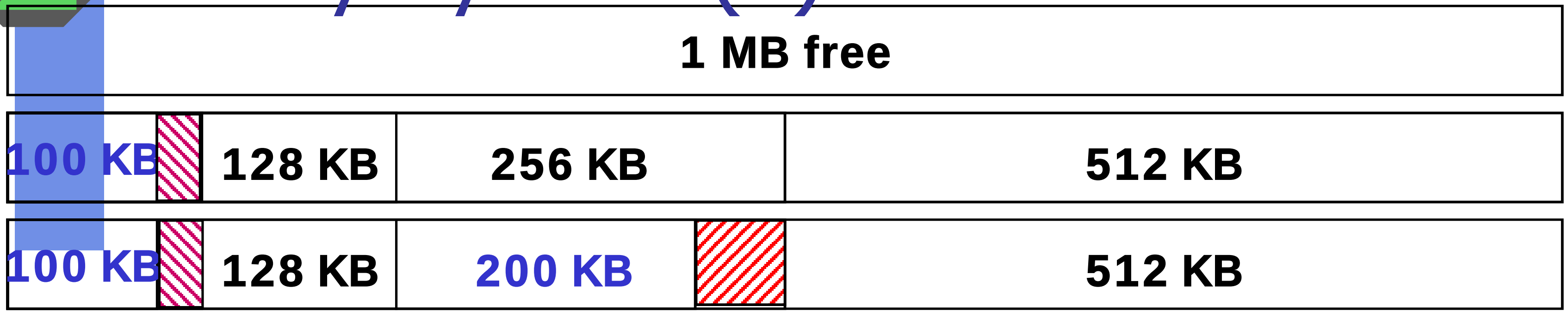


Buddy System (3)



Suppose another 200 KB block is requested

Buddy System (4)

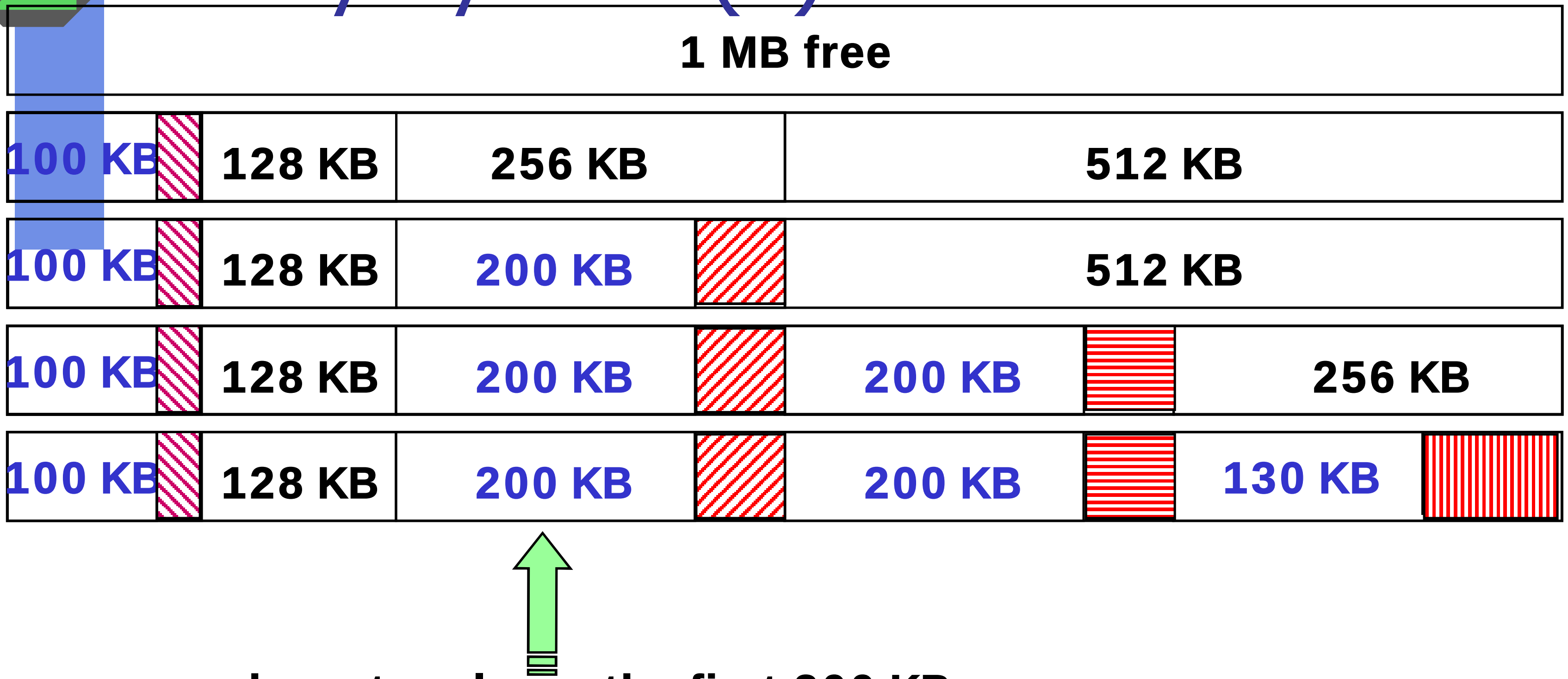


Suppose another 200 KB block is requested

Buddy System (5)

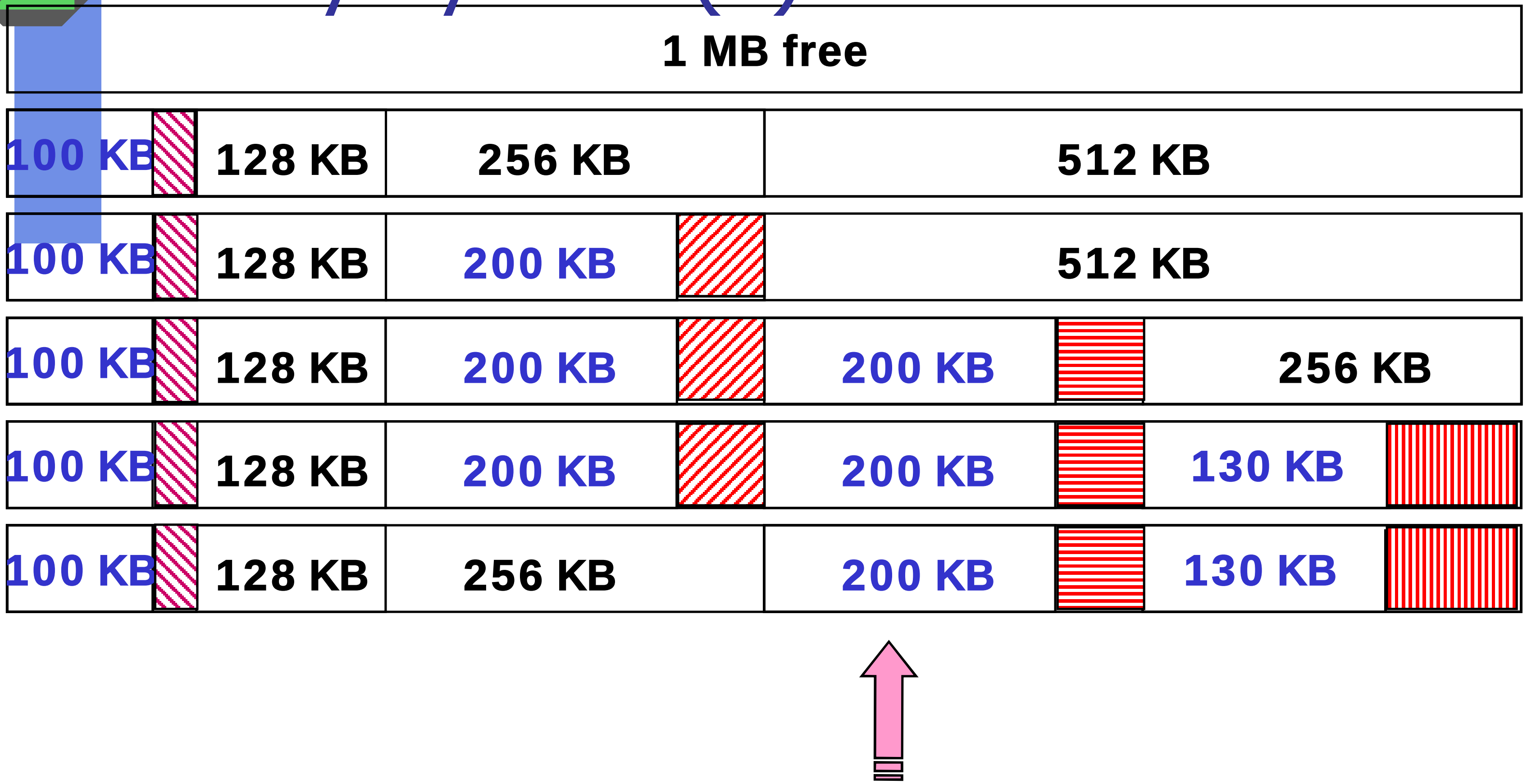
Suppose another 130 KB block is requested

Buddy System (6)



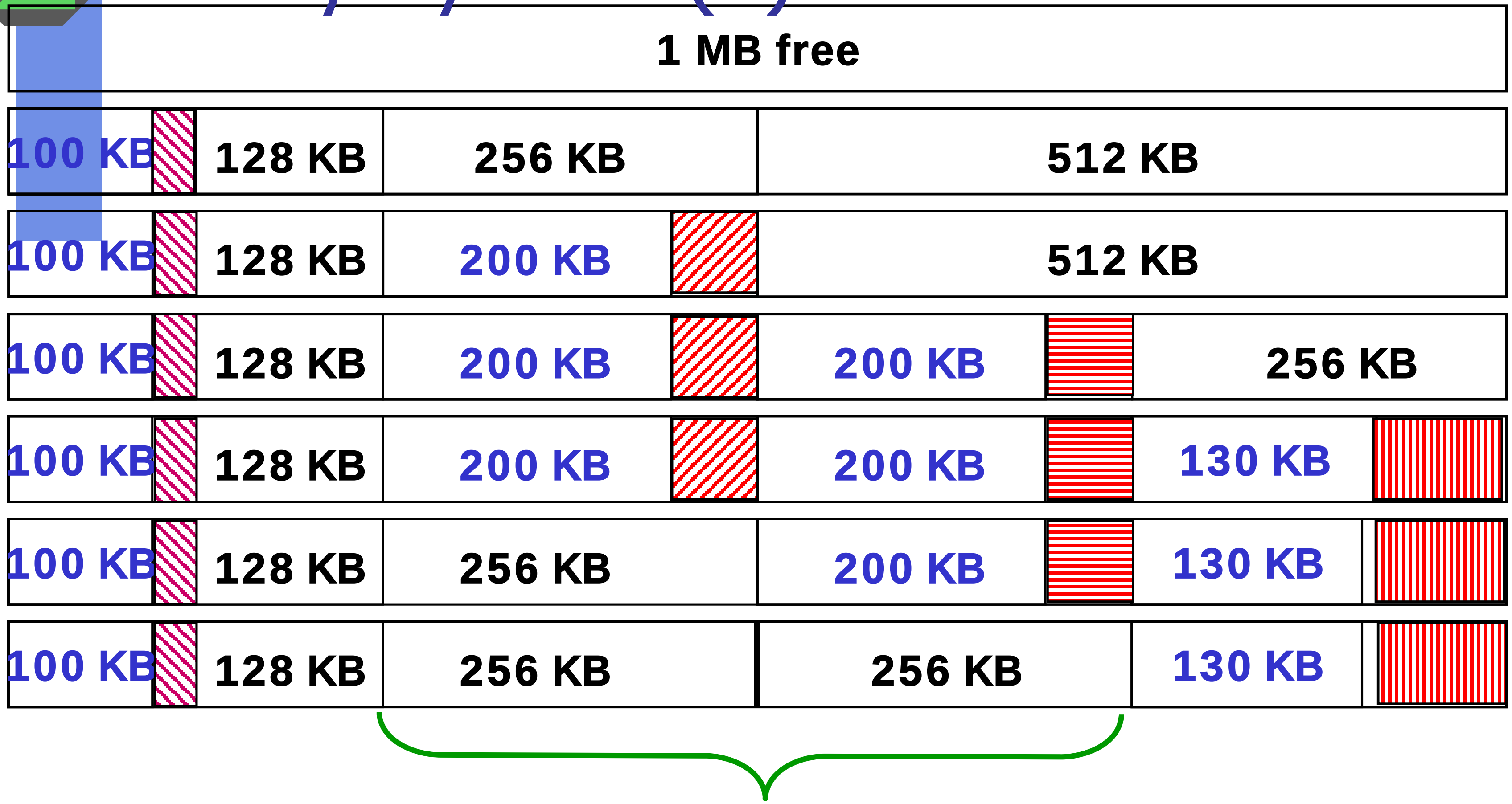
Suppose we have to release the first 200 KB

Buddy System (7)



Suppose we have to release the other 200 KB

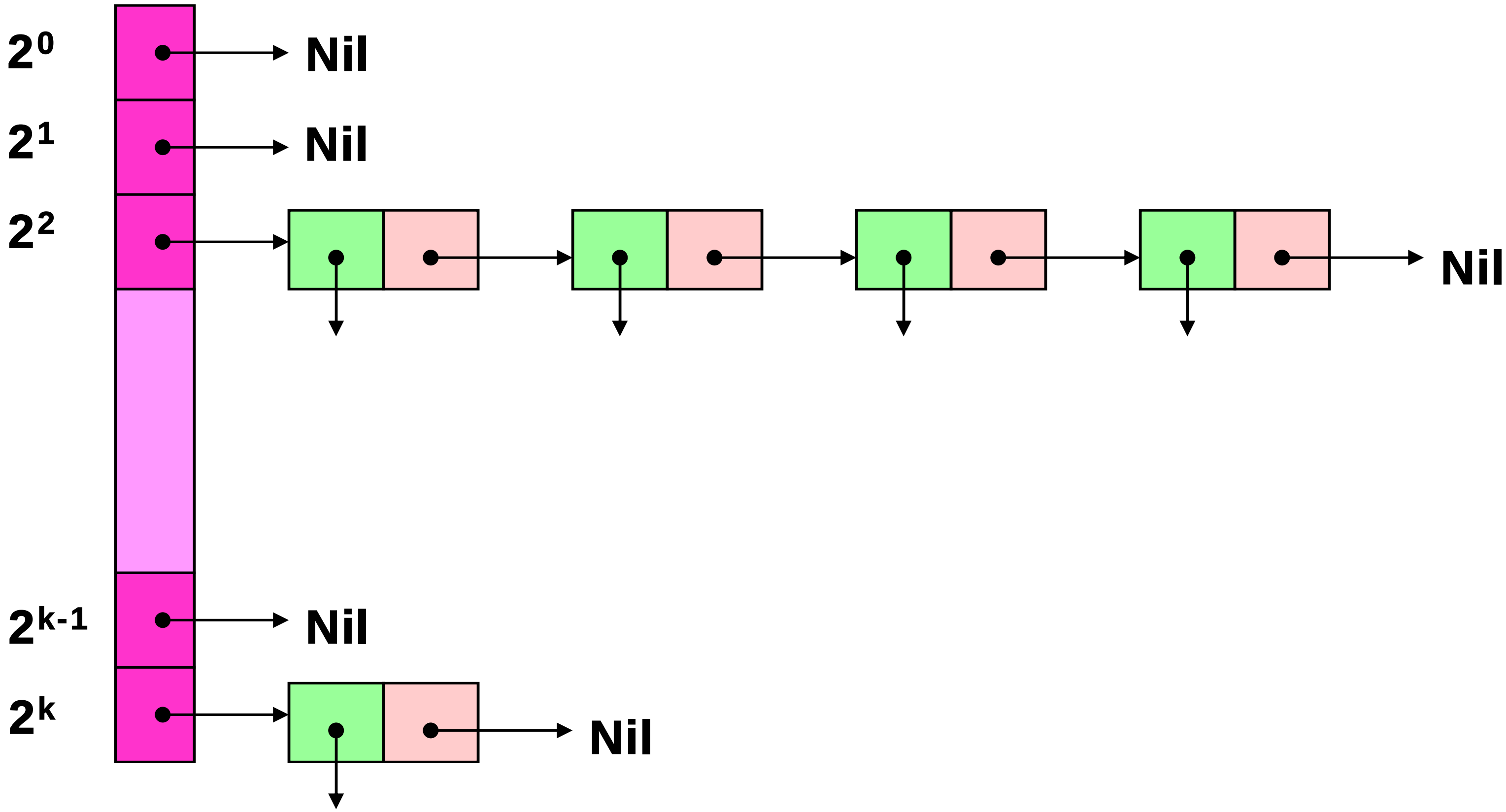
Buddy System (8)



Can we reunify these two blocks forming a 512 KB free block?

Not at all, they are not buddies! Only buddies belong with each other!

Data Structures for Buddy System



Array of heads pointing to free blocks of a certain size 2^k

Operations of Buddy System (1)

Allocating block of size s:

- Round up s to next power of 2, say 2^i ($\Rightarrow$ internal fragmentation)
- Access head of the list for the 2^i pieces
- If list is not empty get first element of list
- If list is empty (recursively) do:
  - Access head of list for the 2^{i+1} pieces
  - If list isn't empty get first element of list
  - cut element in halves
  - take lower half, insert upper half into list for 2^i pieces …

Operations of Buddy System (2)

Releasing a block of size 2^i

- Determine its buddy
- if buddy is (partly) allocated
  insert the block to be released
  into the list of pieces of size 2^i
- if buddy is free reunify both buddies

Question:
How can we efficiently determine the appropriate buddy?

Determining the Buddy

Address calculation for a piece of size 2^{12}

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X	Y	Z	1	0	0	0	0	0	0	0	0	0	0	0	0

address of a piece of size 2^{12}

<u>Question:</u> *What's the address of its buddy?*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X	Y	Z	0	0	0	0	0	0	0	0	0	0	0	0	0

<u>Remark:</u> Other pieces of size 2^{12} differ only in the leading address bits.

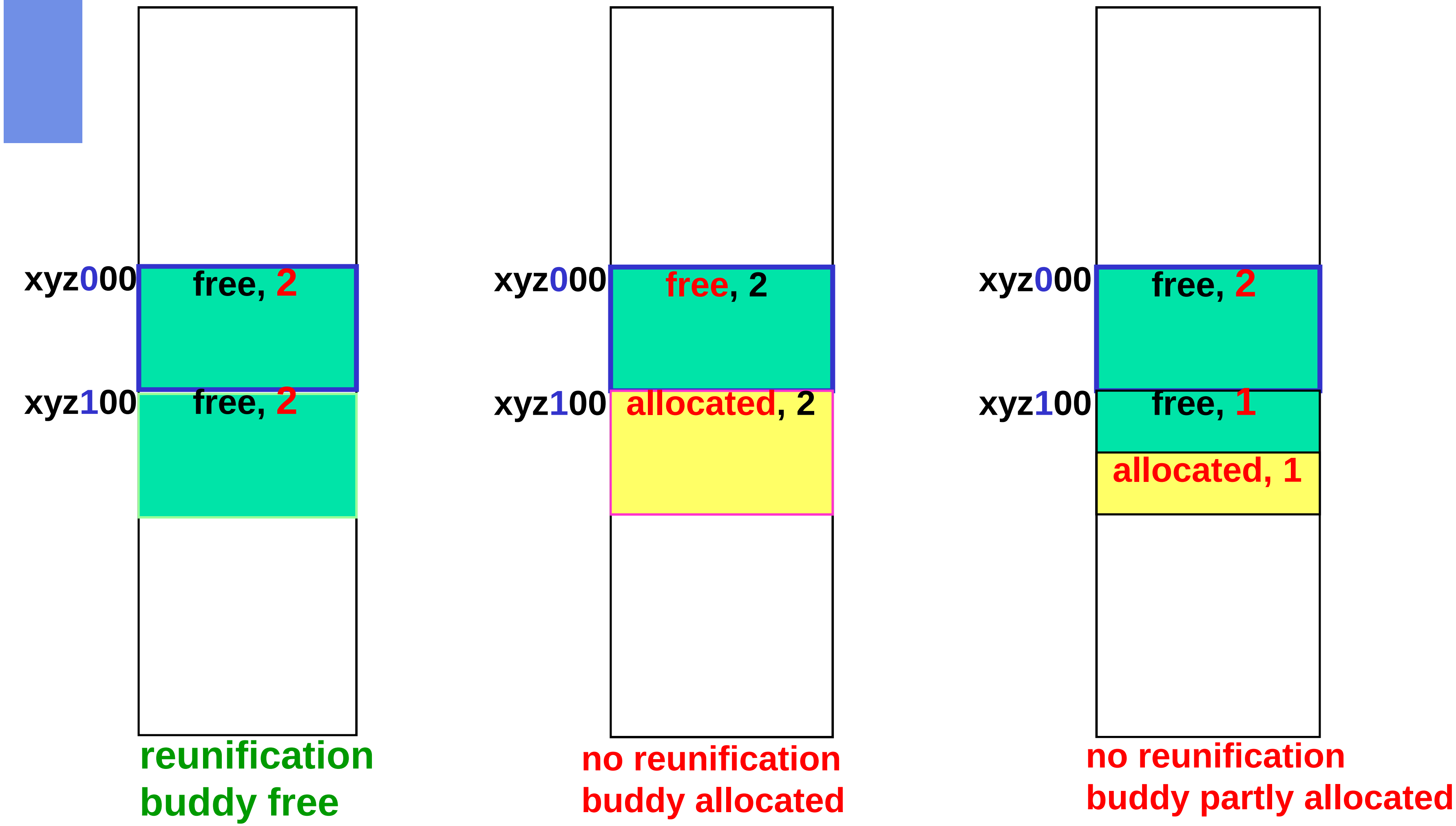
Reunification of Buddies

Each block of size 2^i of buddy system contains:

- State of the block (allocated or free)
- Length of the block

- *Is there an alternative?*
  - Analyze the implementation of Linux
  - Compare both implementations

Scenarios for Reunifications



Summary of Buddy System

On average, the internal fragmentation is about 25%

- each memory block is at least 50% occupied

Works efficiently if the size M of RAM is a power of 2

Slab Allocating (in Linux)

- Inside a kernel there are a few data types, e.g.
  - TCB
  - LAS descriptor
  - Page table
  - File handle …
- Kernel tend to request these data types over and over again, e.g. to establish a new thread/task
- Slab allocation tries to support reusage of previously used data types ⇒ install object type caches
- Following slides:
  - Steffen Wolfer, Proseminar WS 2003 Linux Internals
  - see also: Sven Krohlas, " WS 2004

Slab Allocating (1)

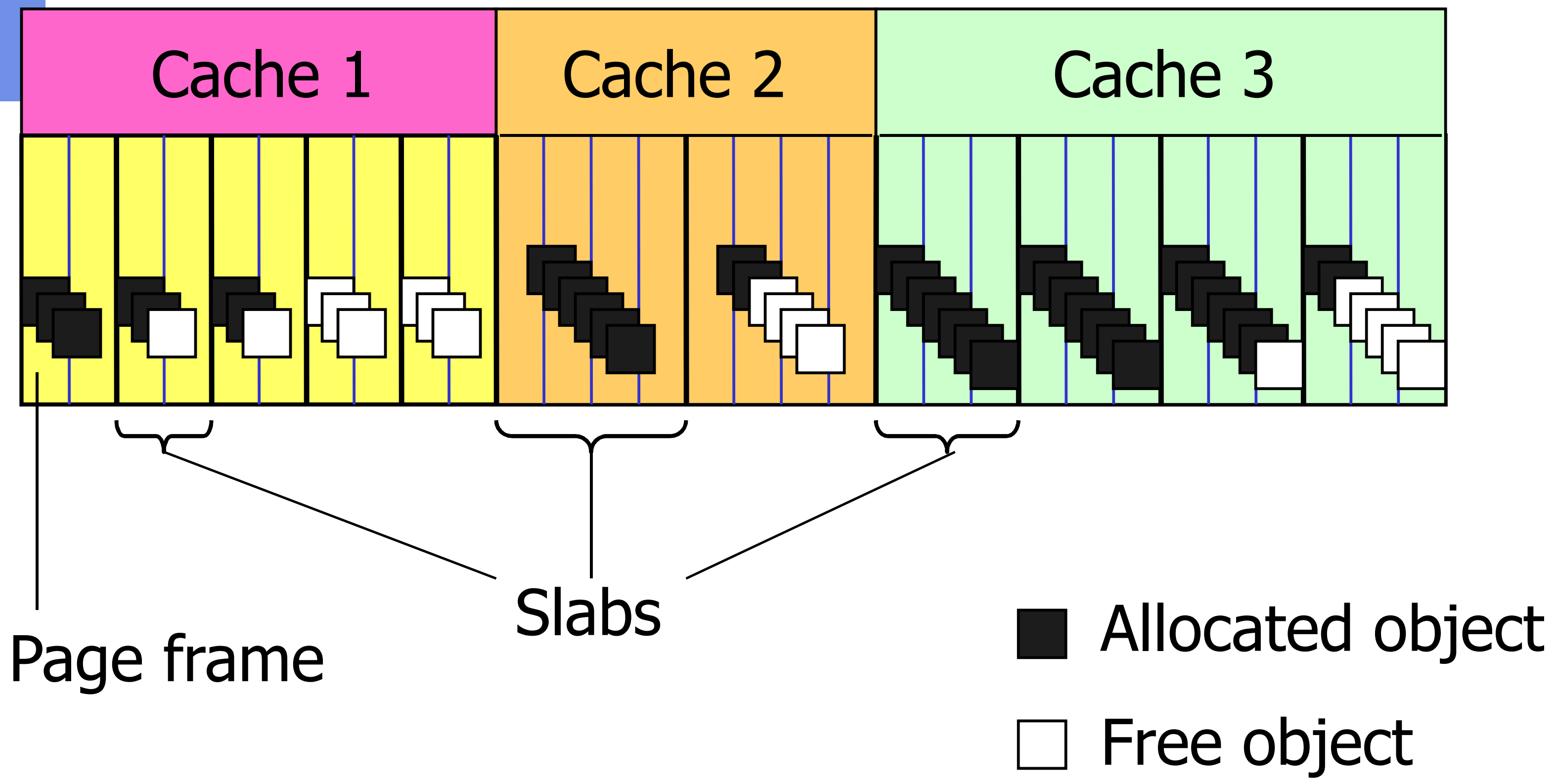
New approach (Sun Microsystems, 1994)

- regard a memory area as an „object container"
- collect objects of the same type in specific "logical kernel caches"
- divide caches into slabs
- slab is part of a specific cache
- slab size = s*page frames, s ≥ 1

Slab Allocating (2)

- Reuse already initialized objects
- Simplify complex allocation
- Take page frames from the buddy system
  - Give back only if buddy system needs them

Slab Allocating (3)



Slab Allocating (4)

Table for the caches:

Cache Descriptor
slabs_full
slabs_partial
slabs_free
next
num
...

- doubled linked lists with slab descriptors for
  - full
  - partially full
  - empty slabs
- pointer to next cache descriptor
- number of objects per slab, size of object, flags, …

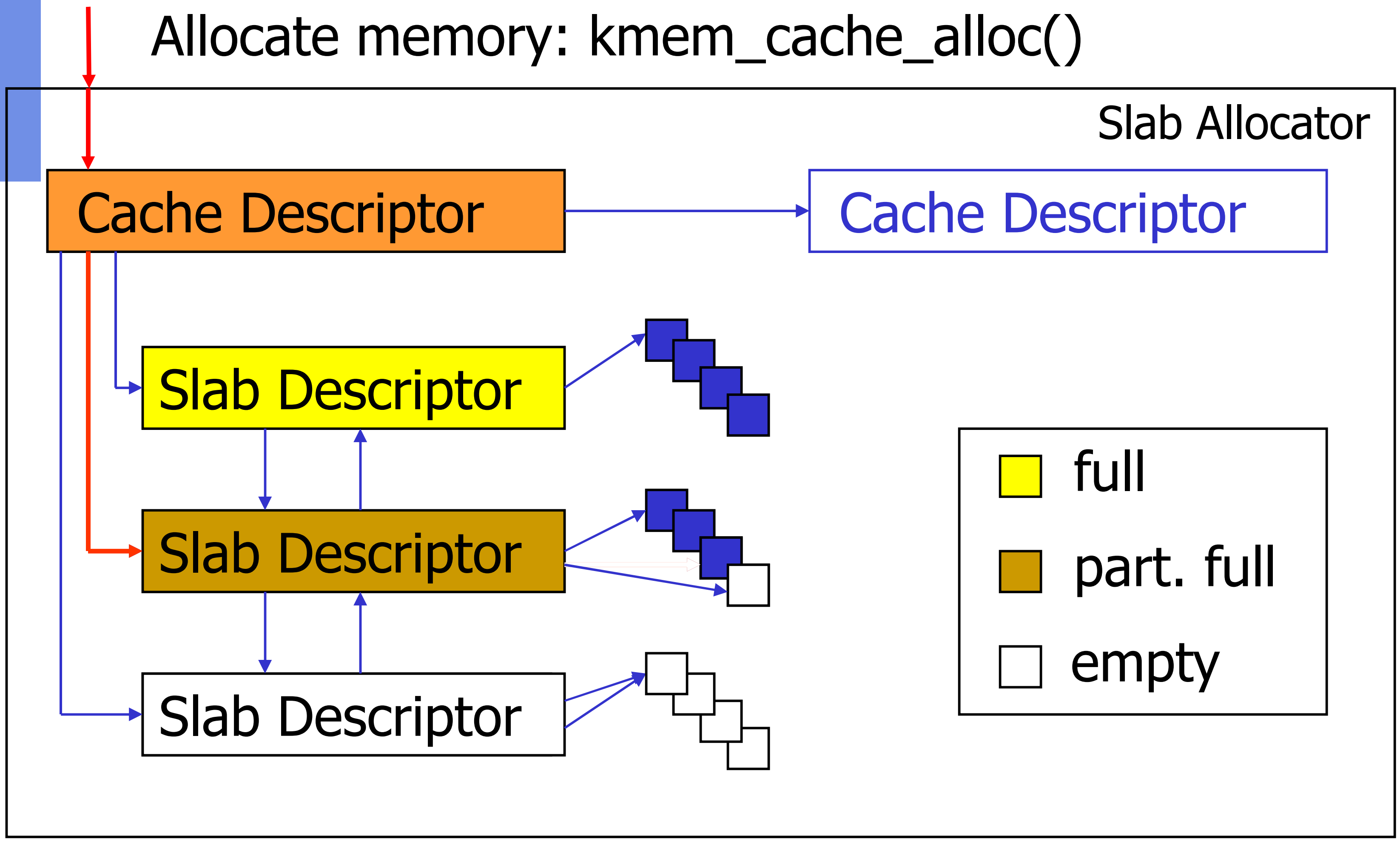
Slab Allocating (5)

Table for Slabs:

Slab Descriptor	
inuse	number of currently allocated objects
s_mem	pointer to 1. Object
free	pointer to 1. free Object
list	pointer to list of slab descriptors
...	

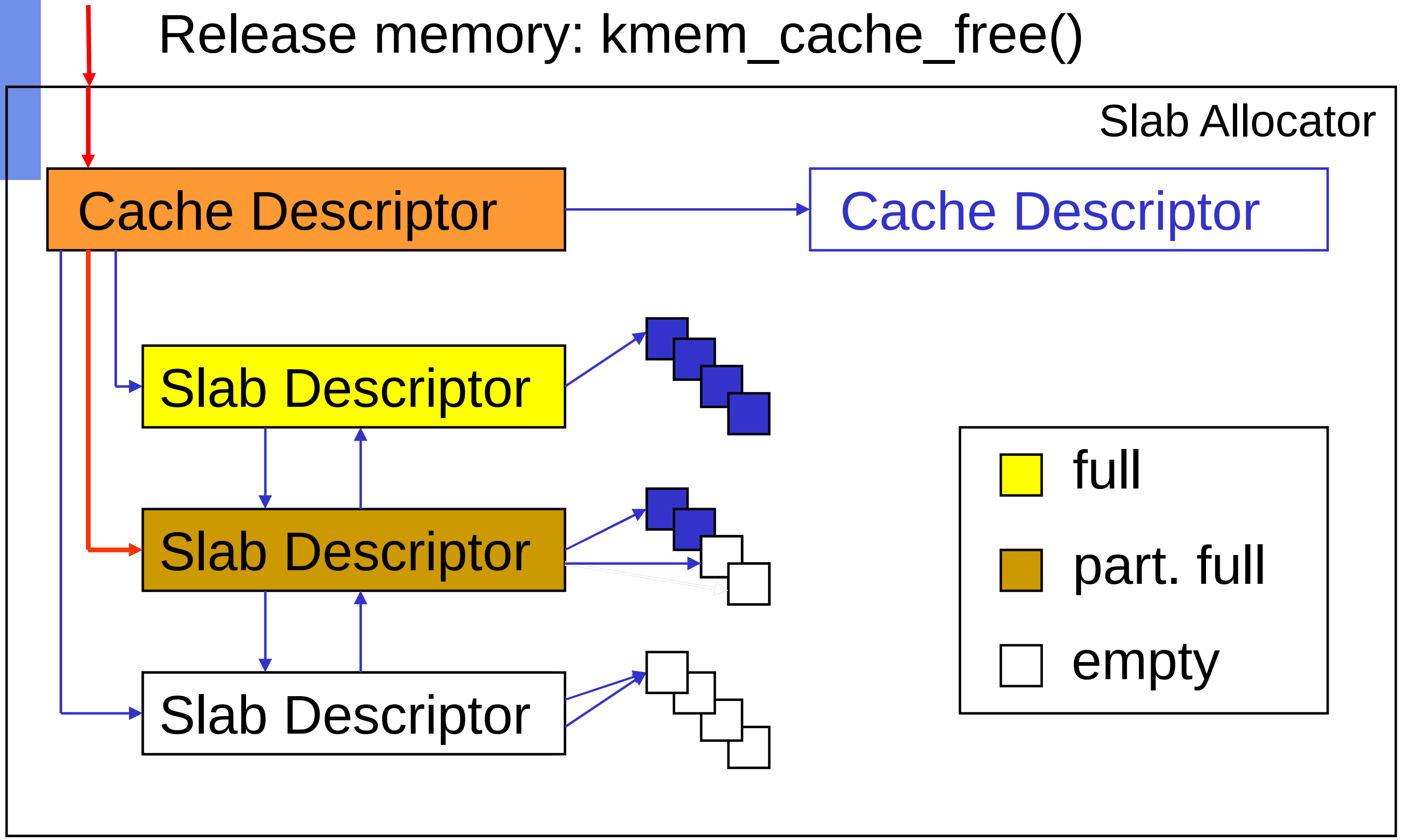
Slab Allocating (6)

Allocate memory: kmem_cache_alloc()



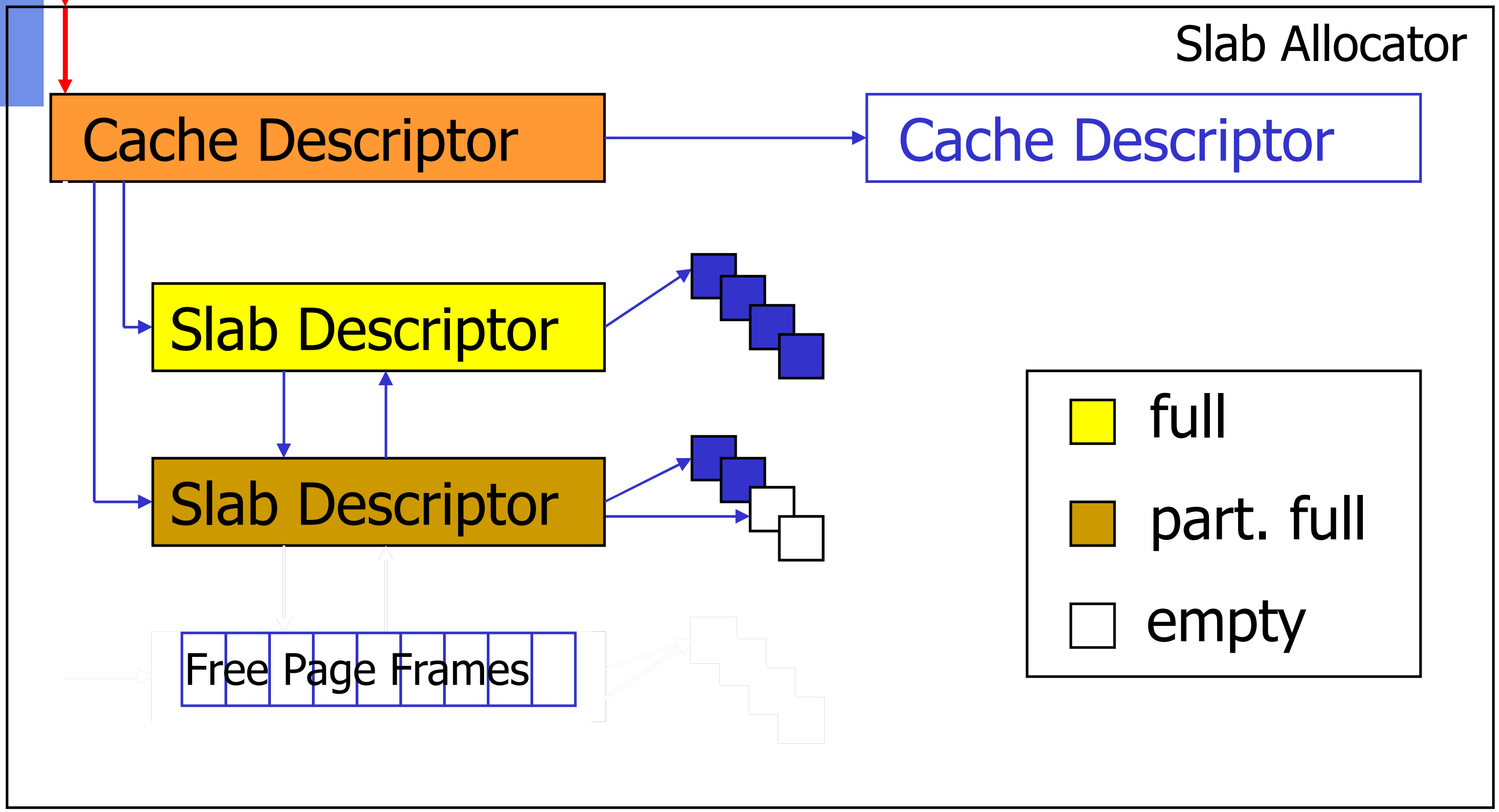
Slab Allocating (7)

Release memory: kmem_cache_free()



Slab Allocating (8)

- Free slabs: kmem_cache_destroy()



Summary of Slab Allocation

- *Who is using slab allocation?*
- *What kernel data types are mapped to slabs?*
- *How much space in total is managed by slabs?*
- *Do slabs have to be mapped to contiguous memory?*
- *...*