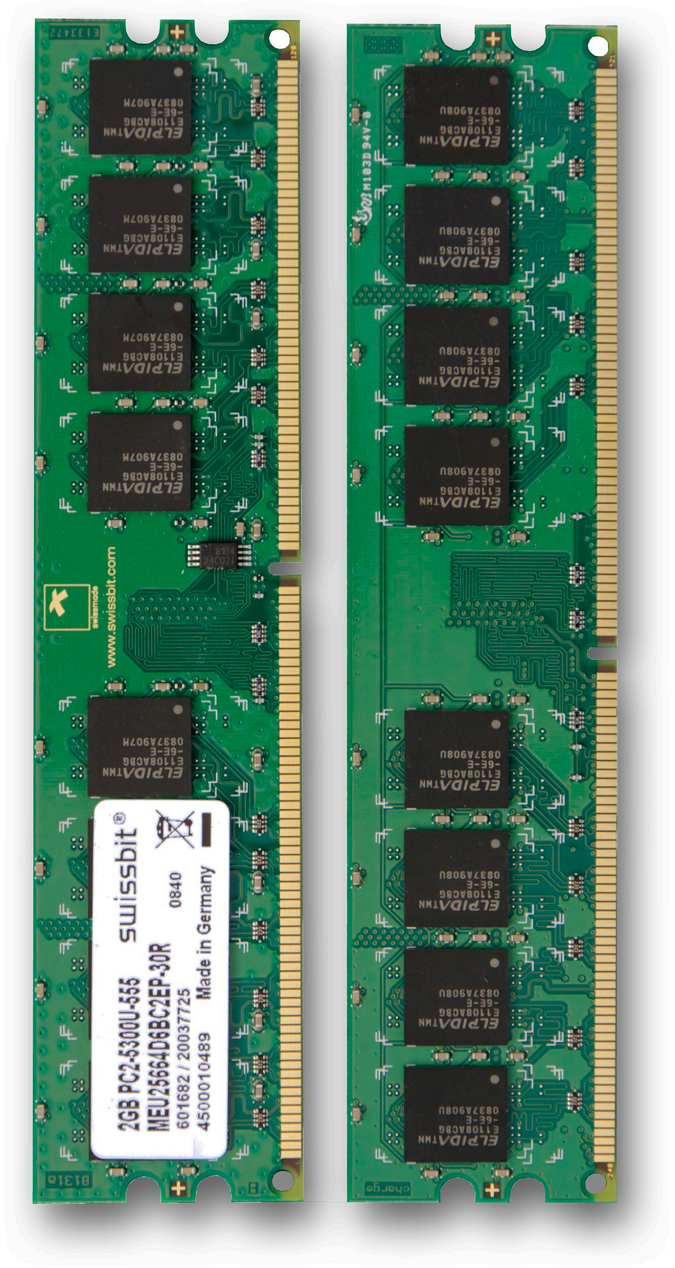
Operating Systems

Lecture 16: Swapping + Free Memory

Nipun Batra

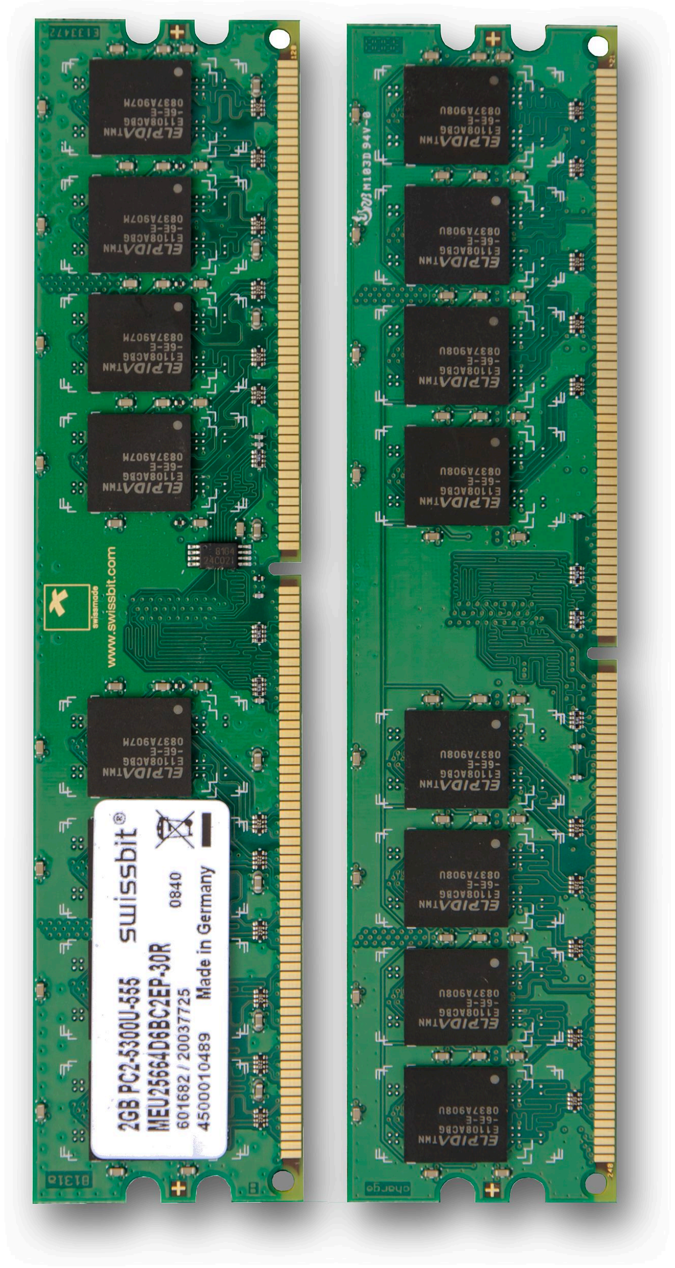
Sep 11, 2018

Swapping

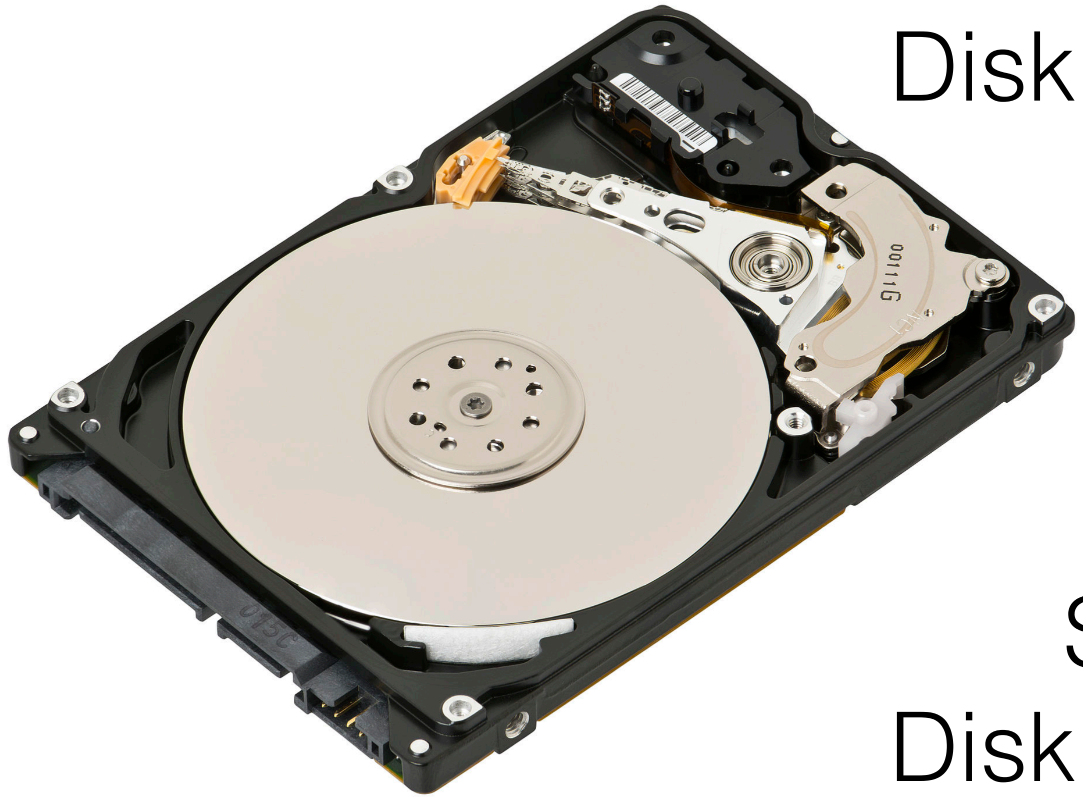


Swapping

Swap in
Disk —> Memory

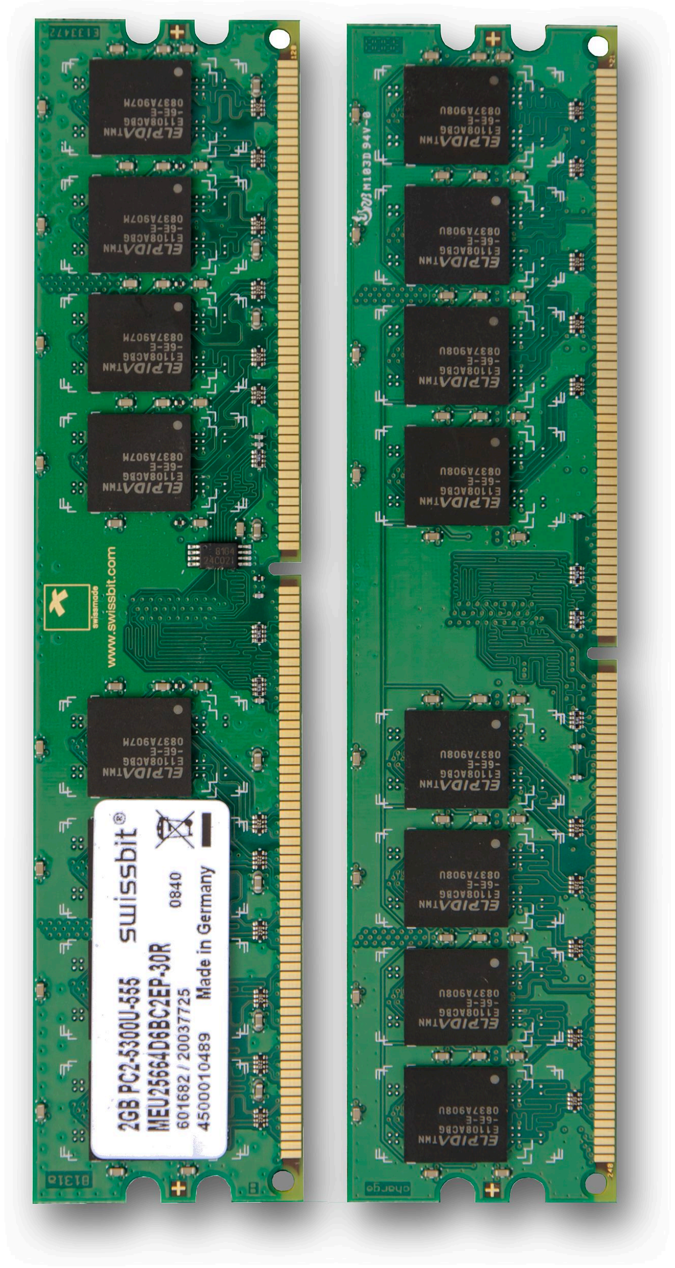


Swapping

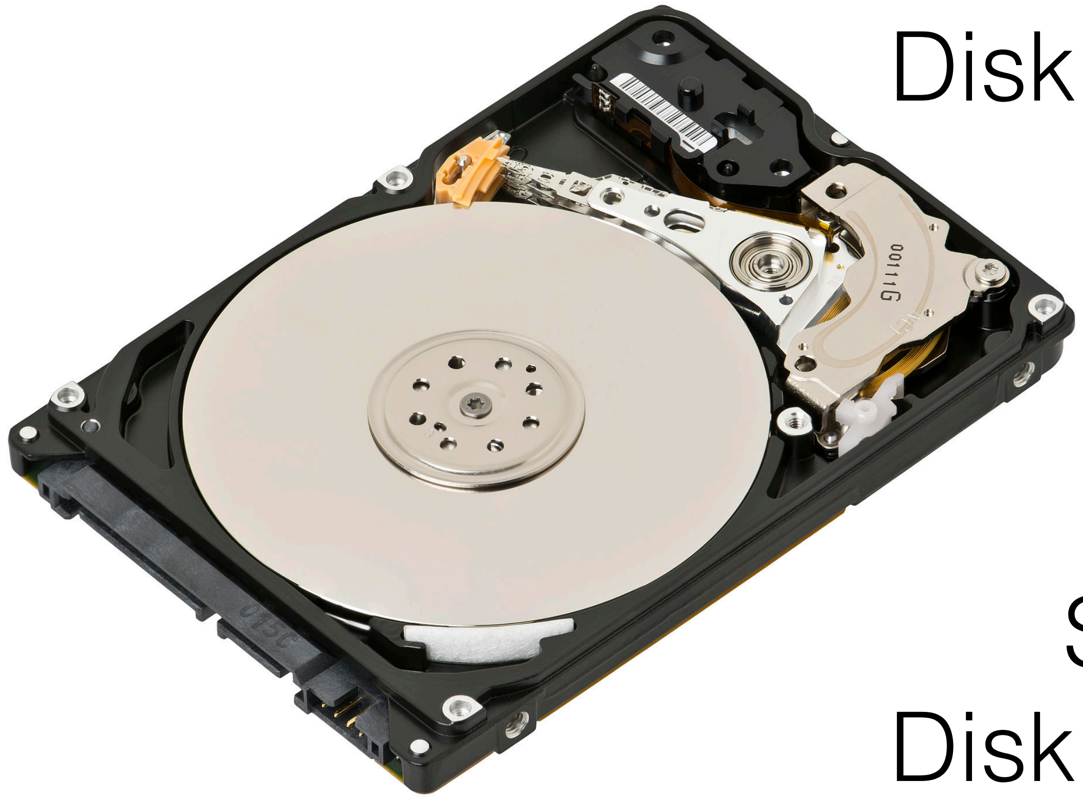


Swap in
Disk —> Memory

Swap out
Disk <— Memory

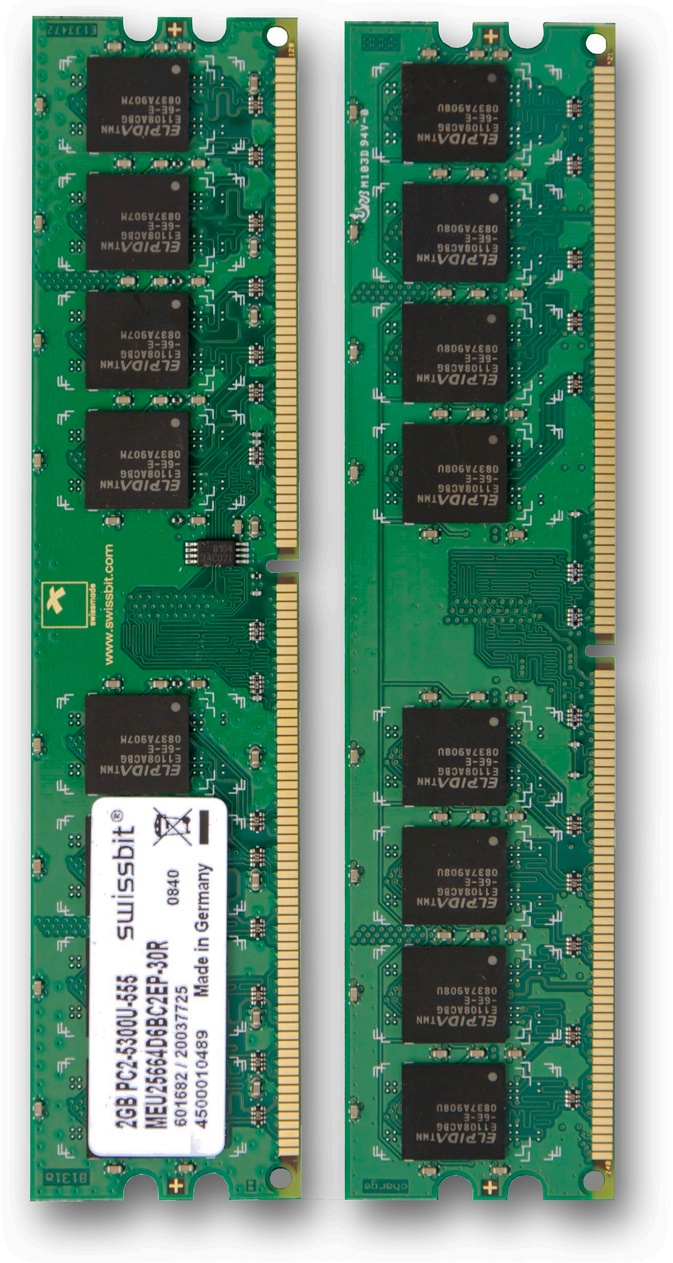


Swapping



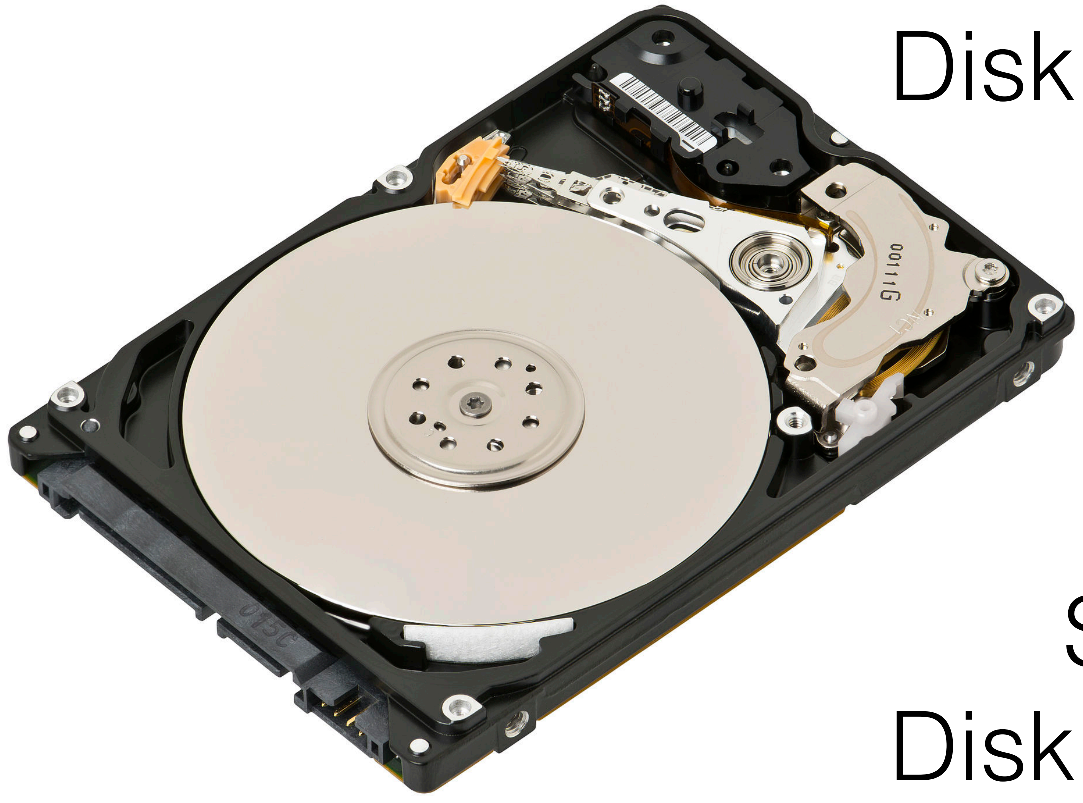
Swap in
Disk —> Memory

Swap out
Disk <— Memory



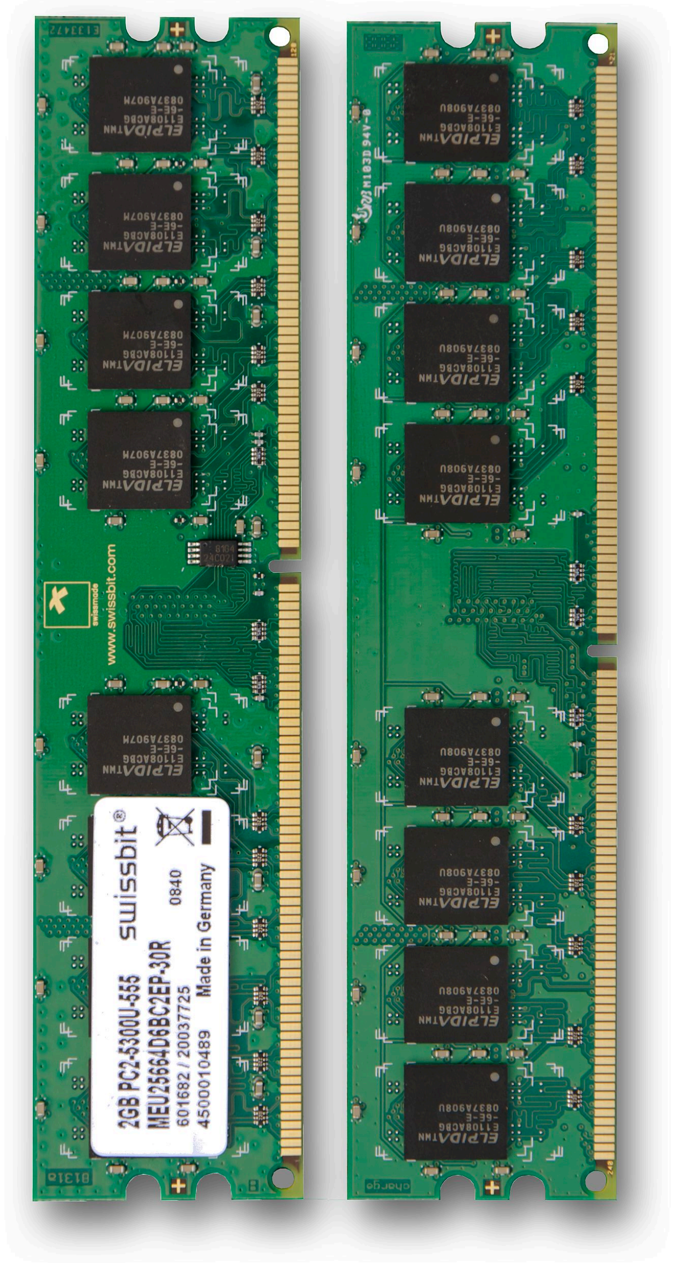
Done well : Memory as large as disk, as fast as RAM

Swapping



Swap in
Disk —> Memory

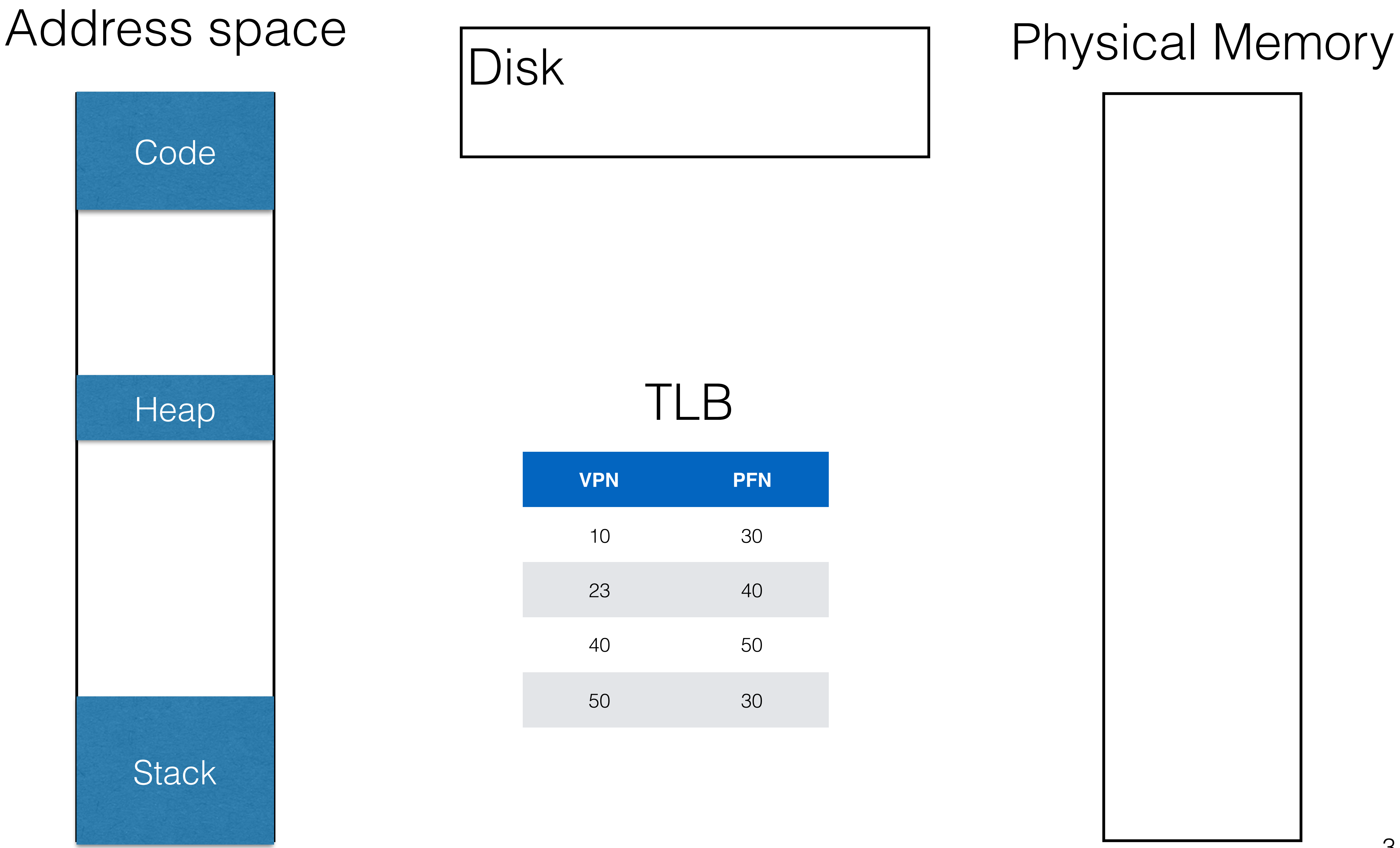
Swap out
Disk <— Memory



Done well : Memory as large as disk, as fast as RAM

Done bad : Memory as small as RAM, as slow as disk

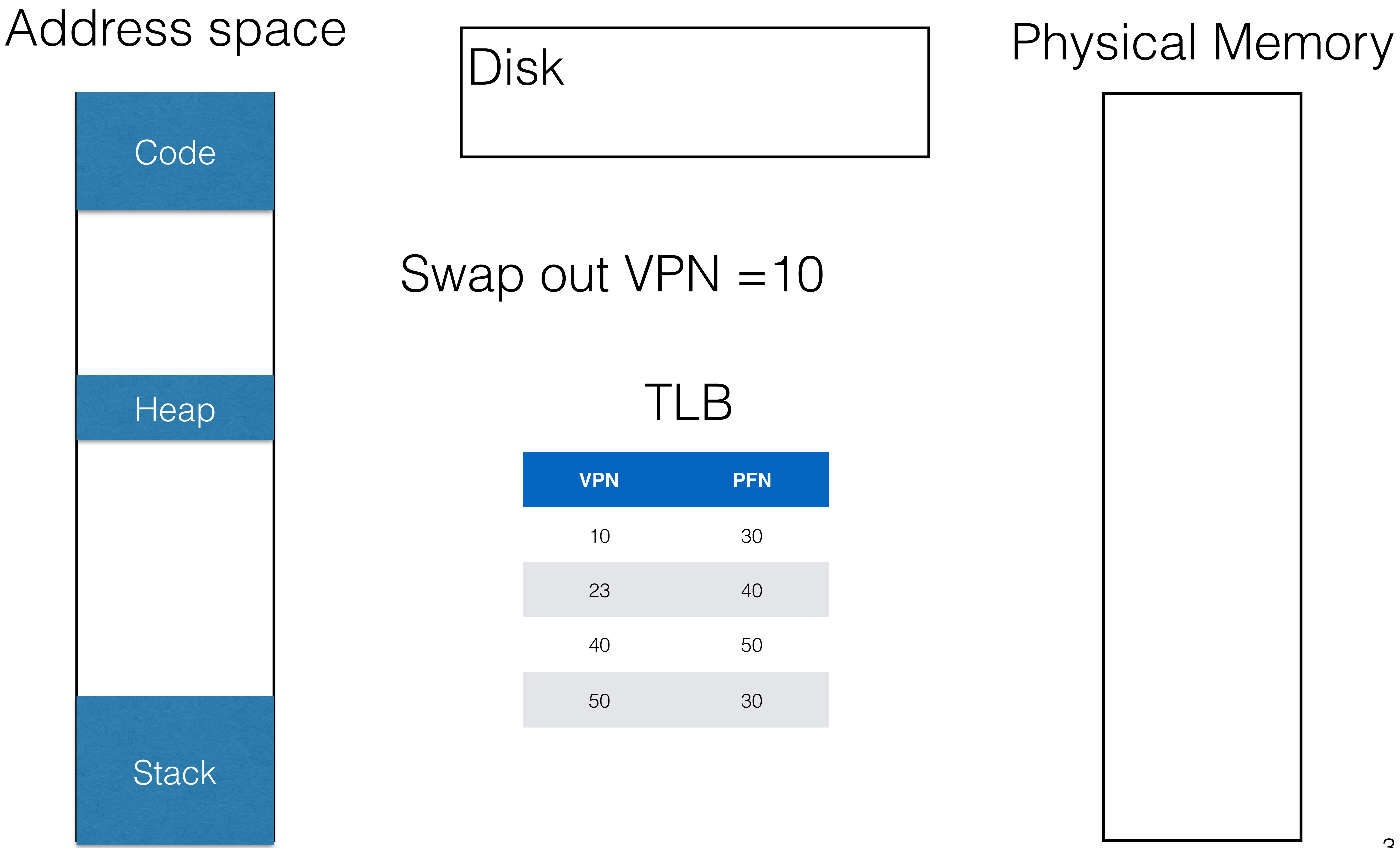
Swapping Out



TLB

VPN	PFN
10	30
23	40
40	50
50	30

Swapping Out

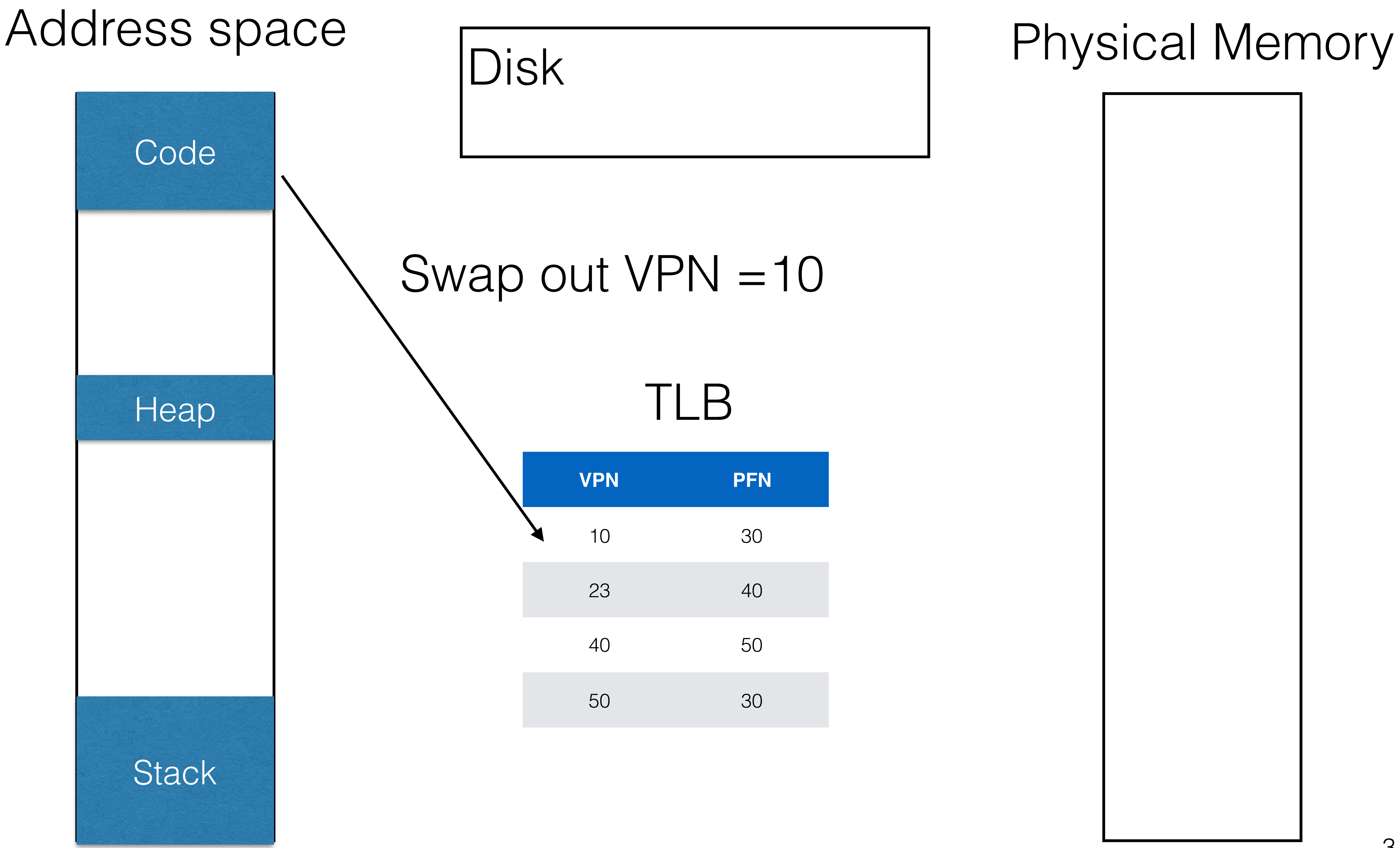


Swap out VPN =10

TLB

VPN	PFN
10	30
23	40
40	50
50	30

Swapping Out

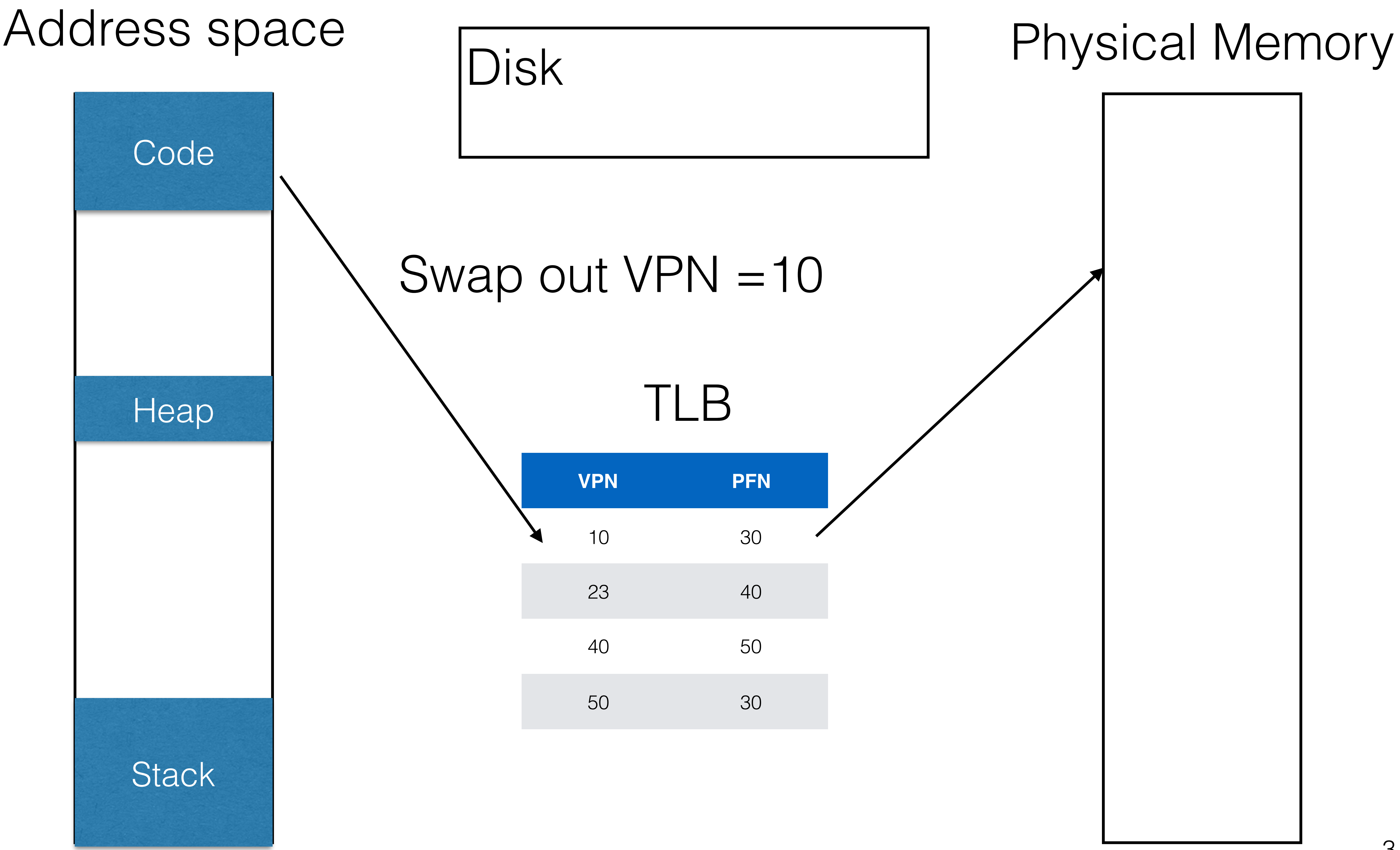


Swap out VPN =10

TLB

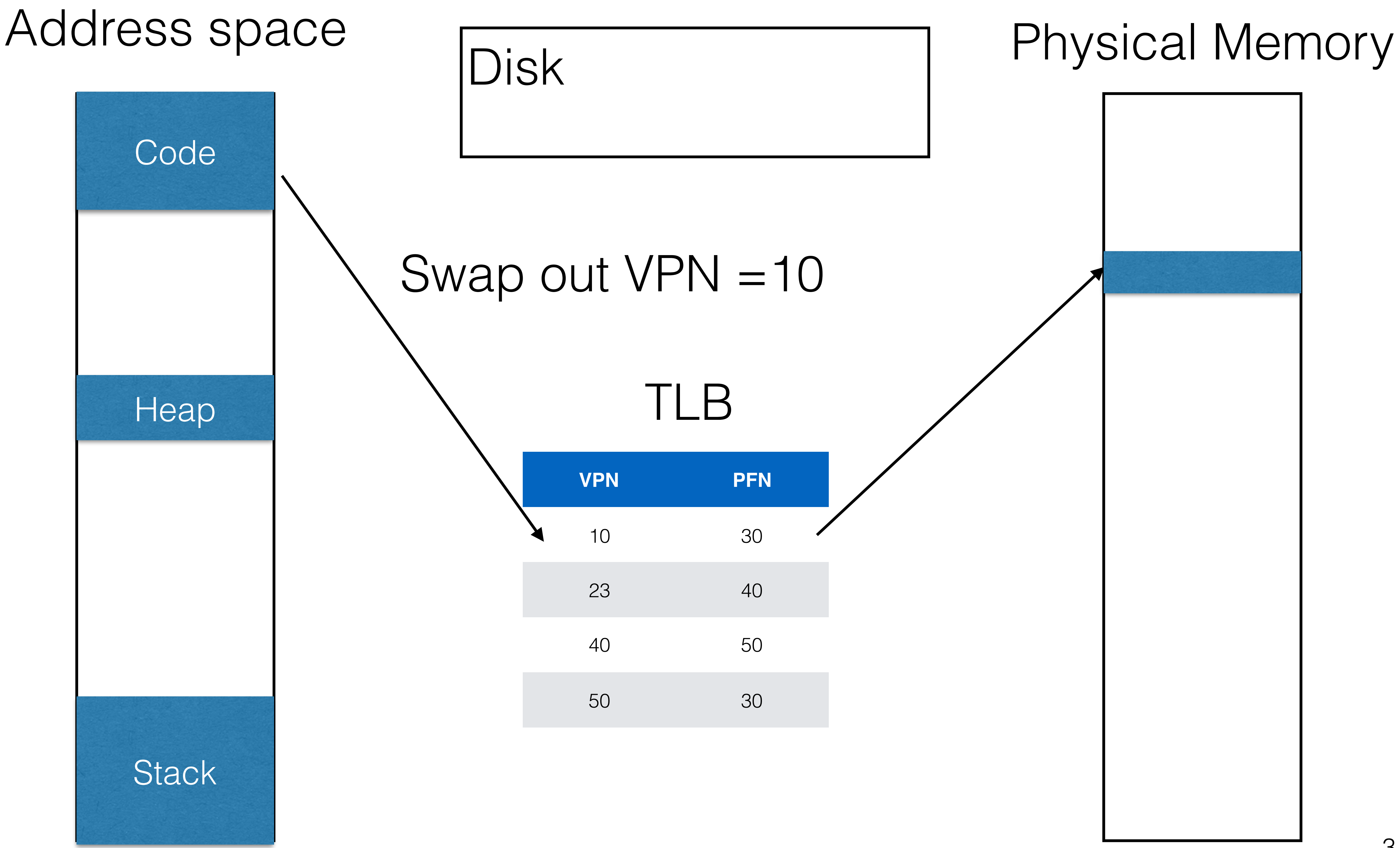
VPN	PFN
10	30
23	40
40	50
50	30

Swapping Out



VPN	PFN
10	30
23	40
40	50
50	30

Swapping Out

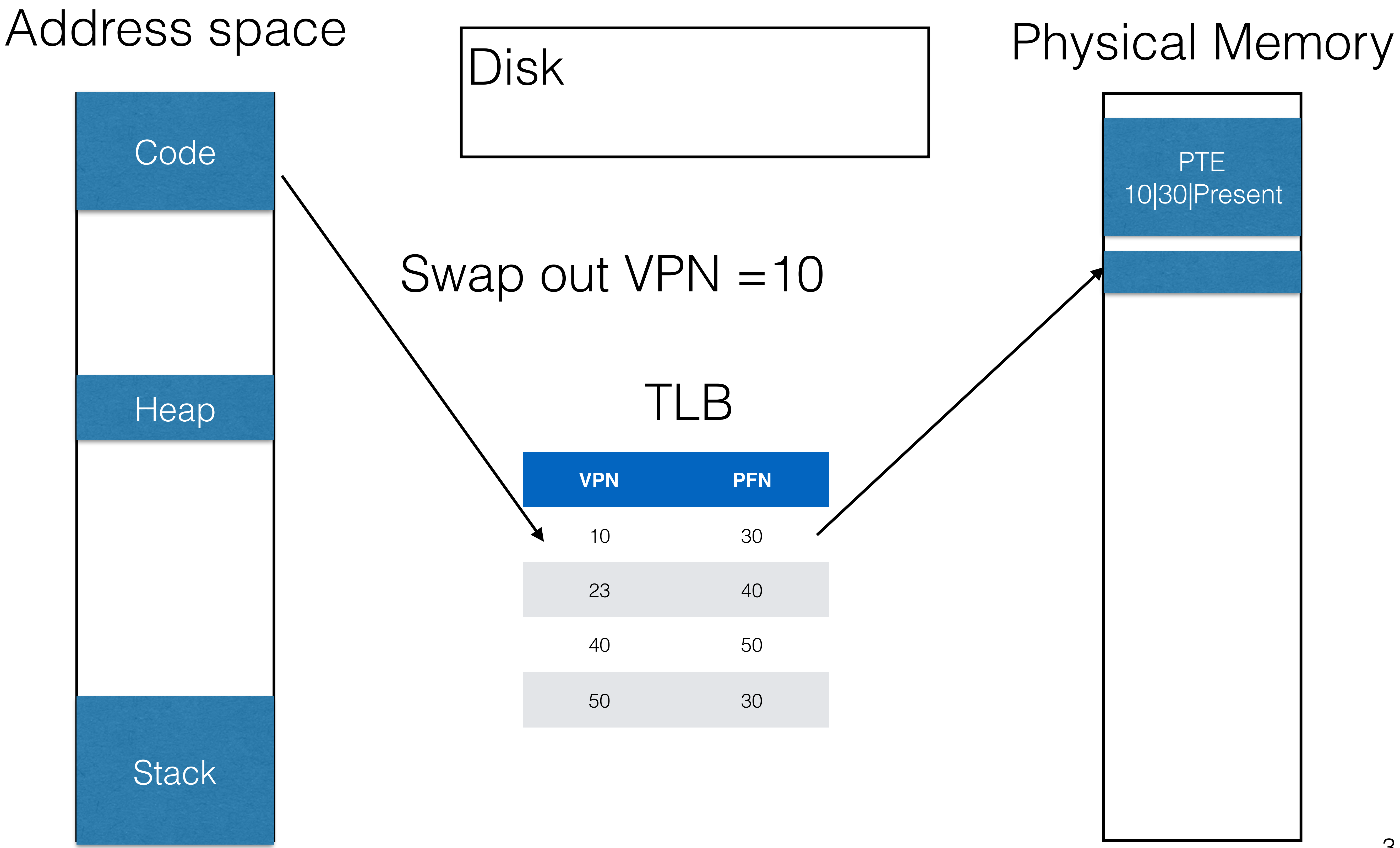


Swap out VPN =10

TLB

VPN	PFN
10	30
23	40
40	50
50	30

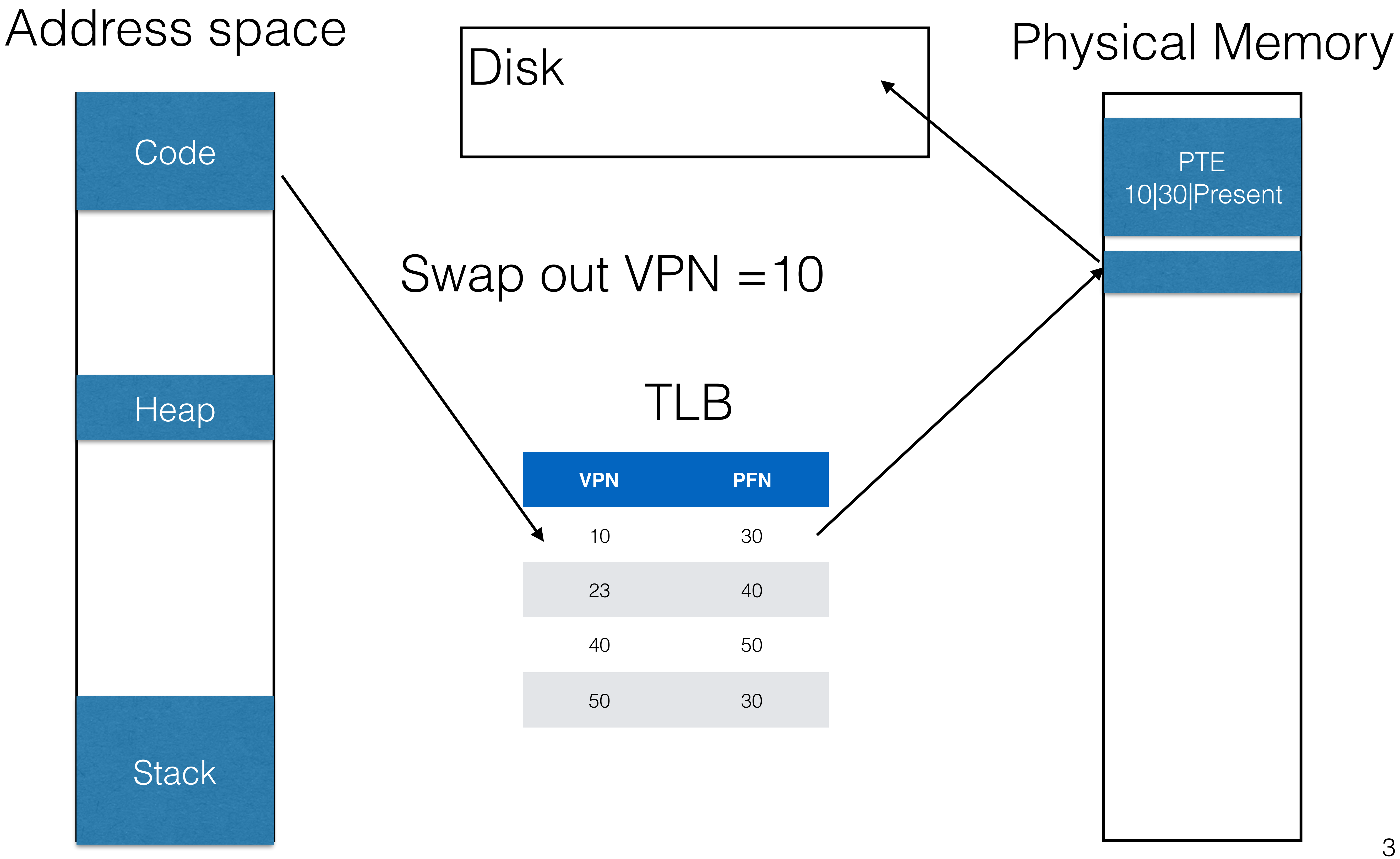
Swapping Out



Swap out VPN =10

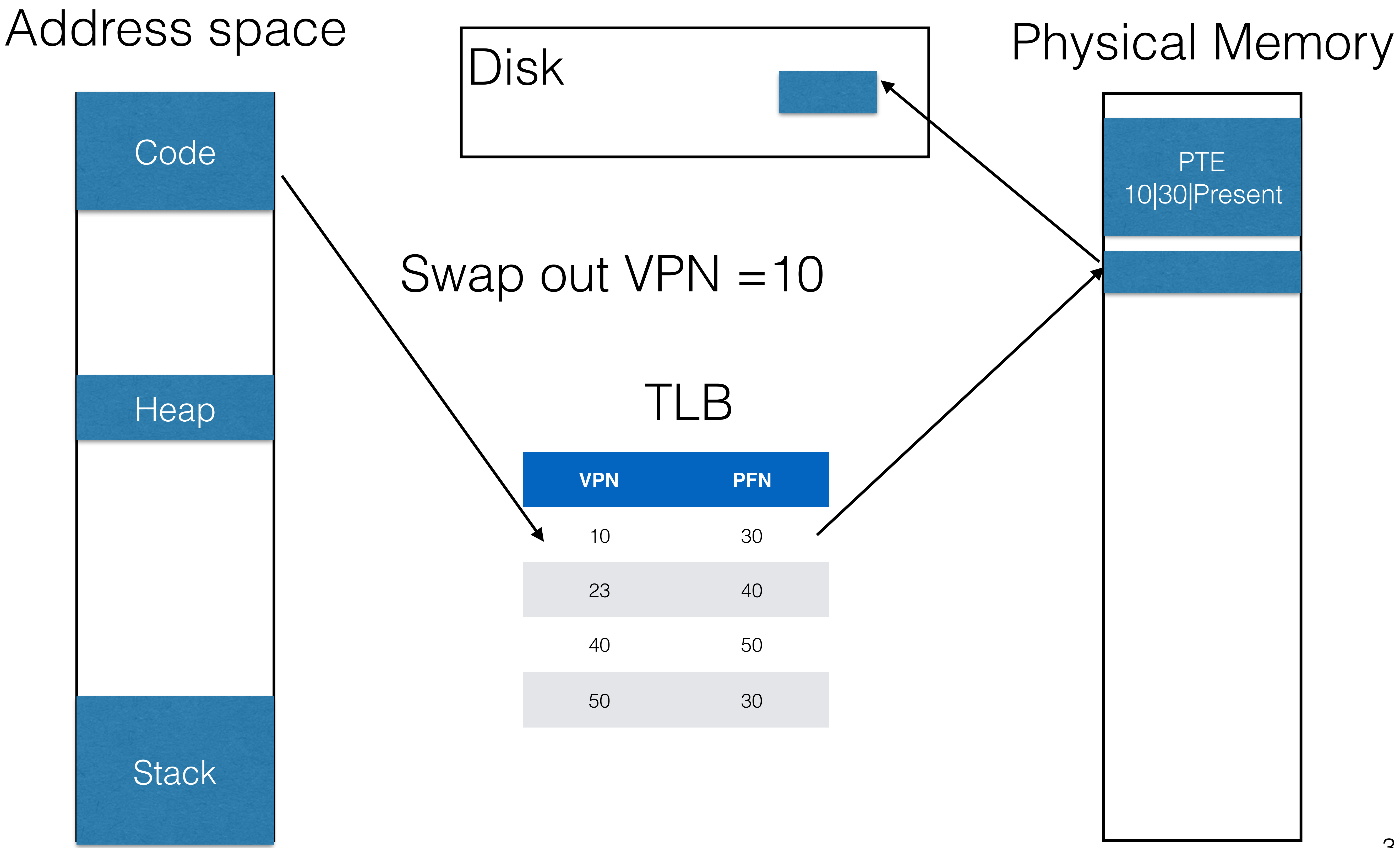
VPN	PFN
10	30
23	40
40	50
50	30

Swapping Out



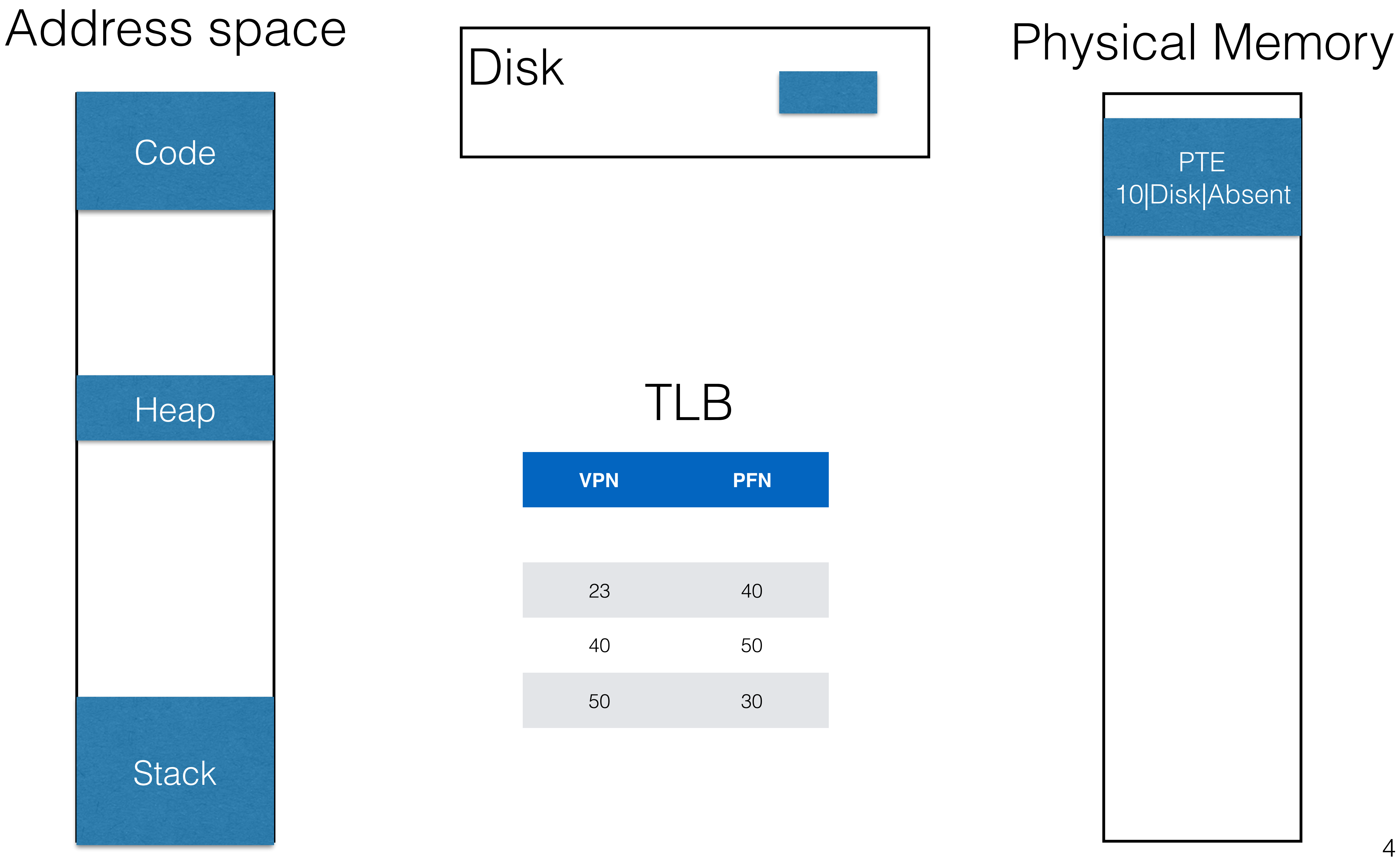
VPN	PFN
10	30
23	40
40	50
50	30

Swapping Out



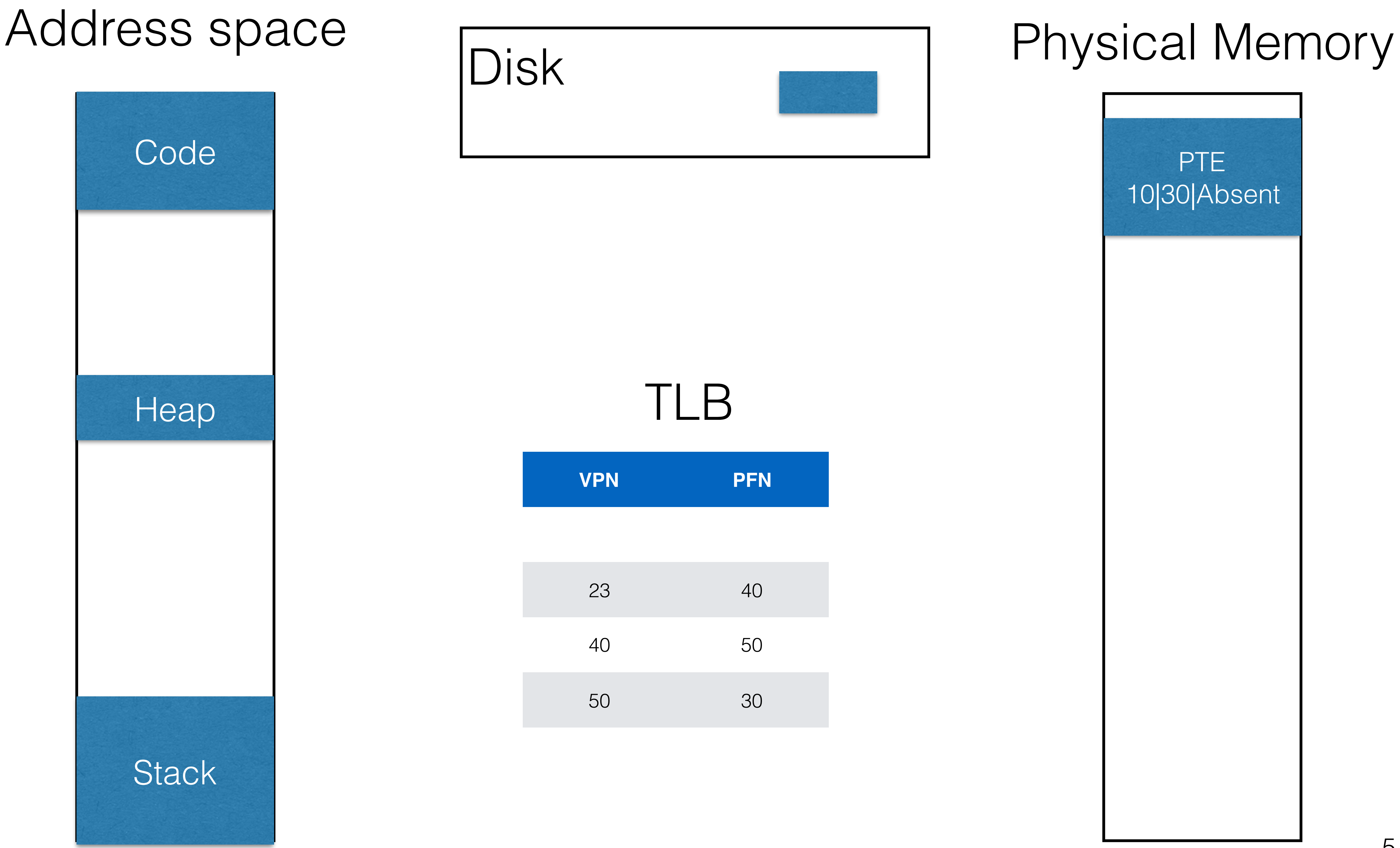
VPN	PFN
10	30
23	40
40	50
50	30

Swapping Out



VPN	PFN
23	40
40	50
50	30

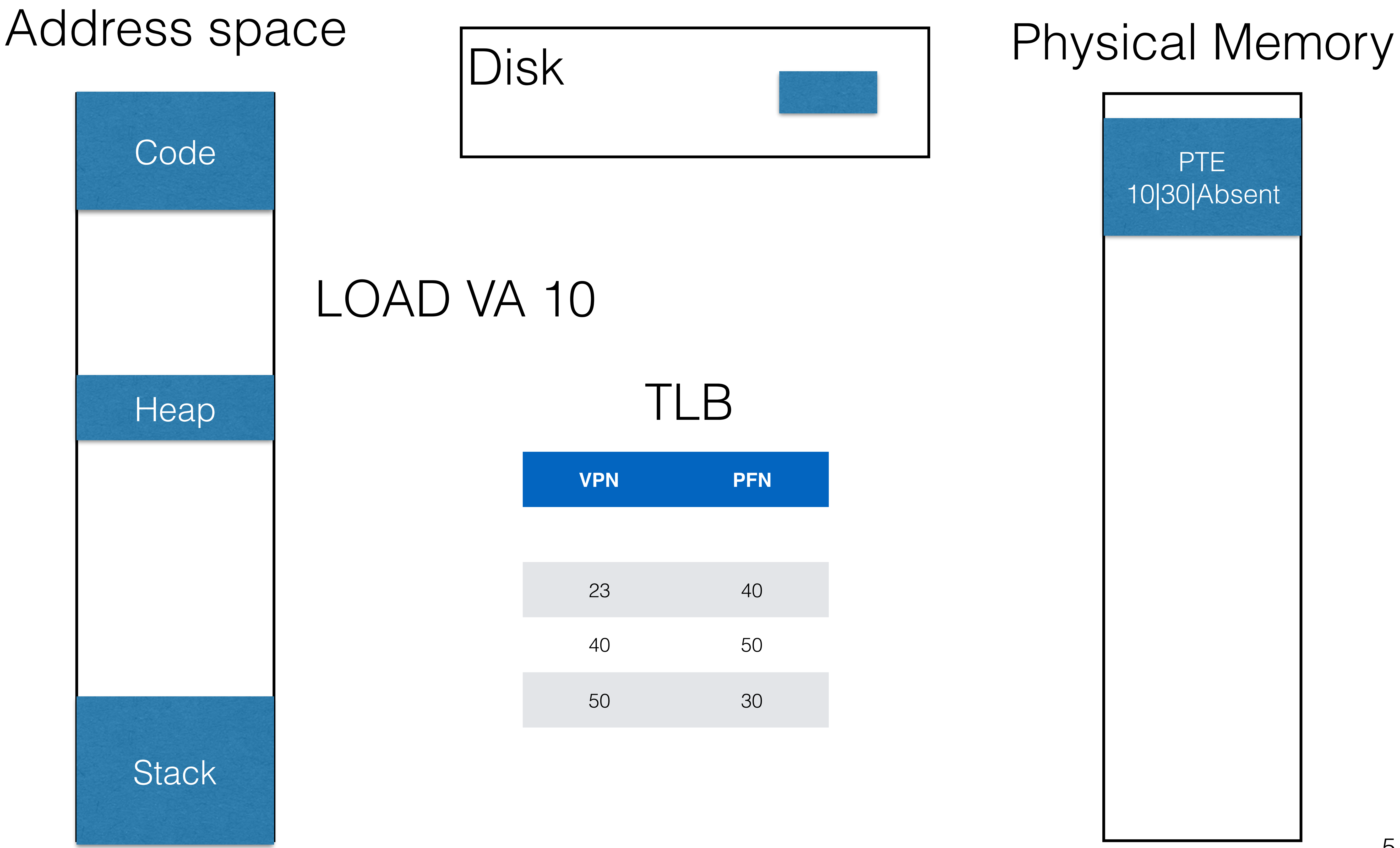
Swapping in



TLB

VPN	PFN
23	40
40	50
50	30

Swapping in

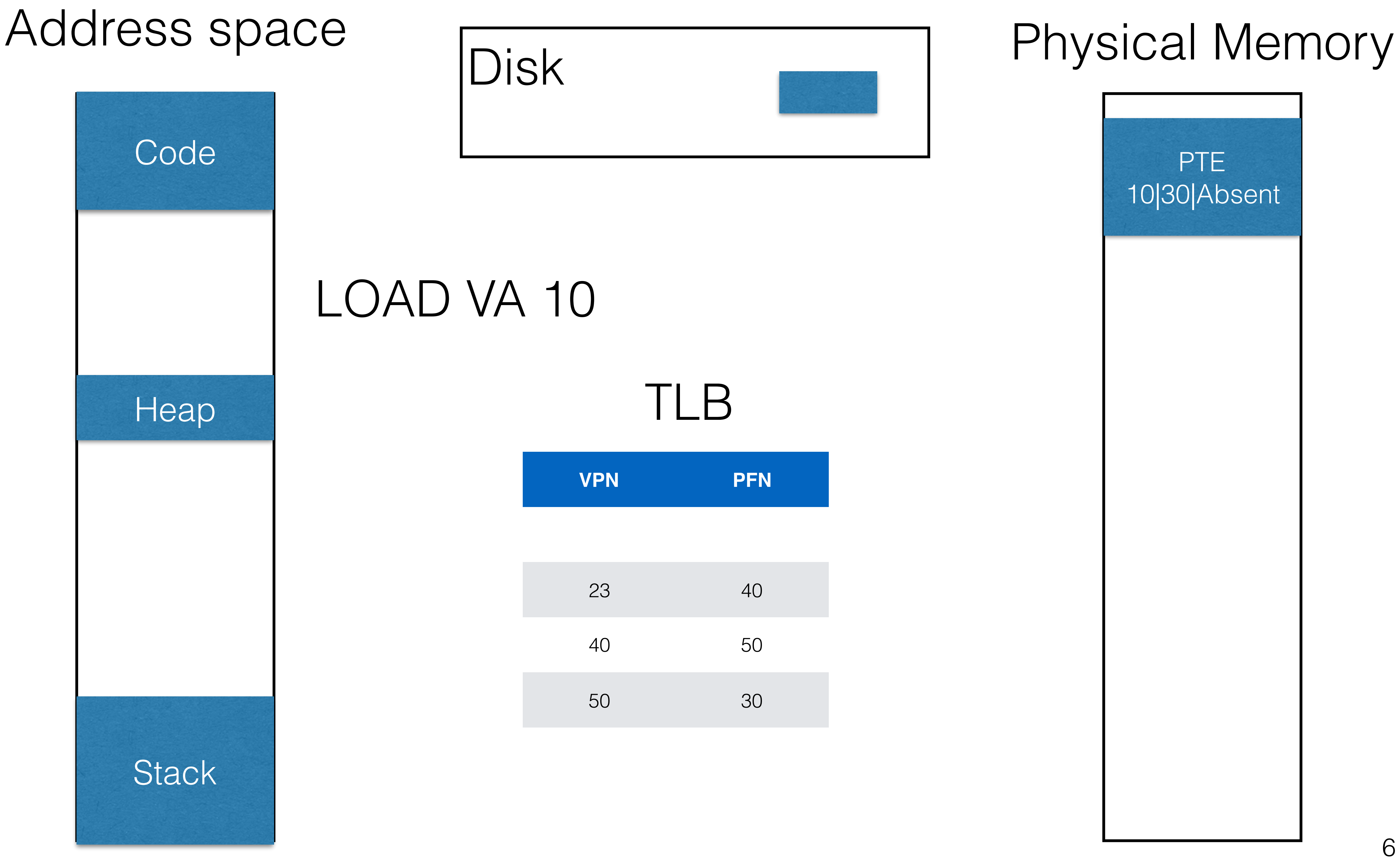


LOAD VA 10

TLB

VPN	PFN
23	40
40	50
50	30

Swapping in

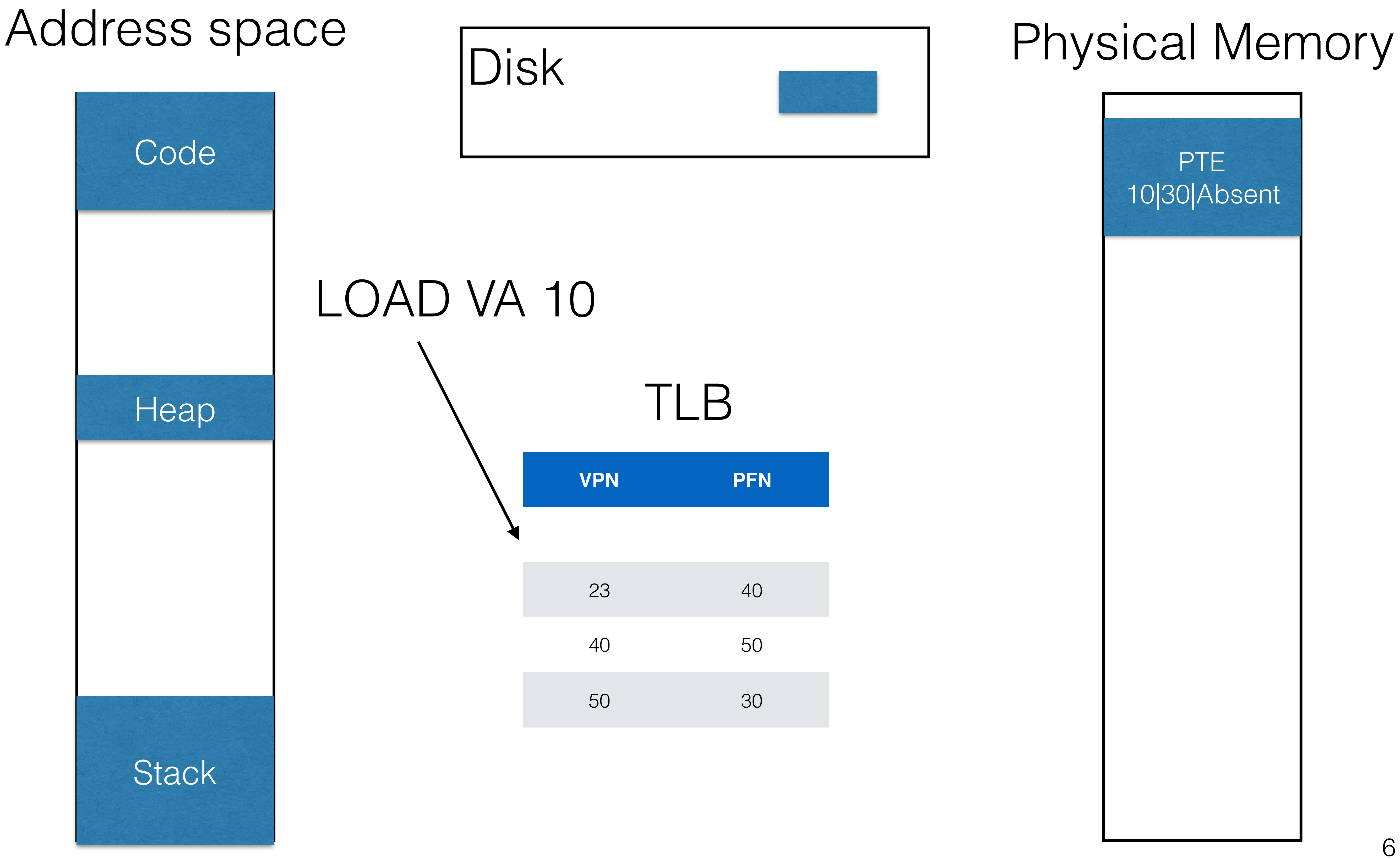


LOAD VA 10

TLB

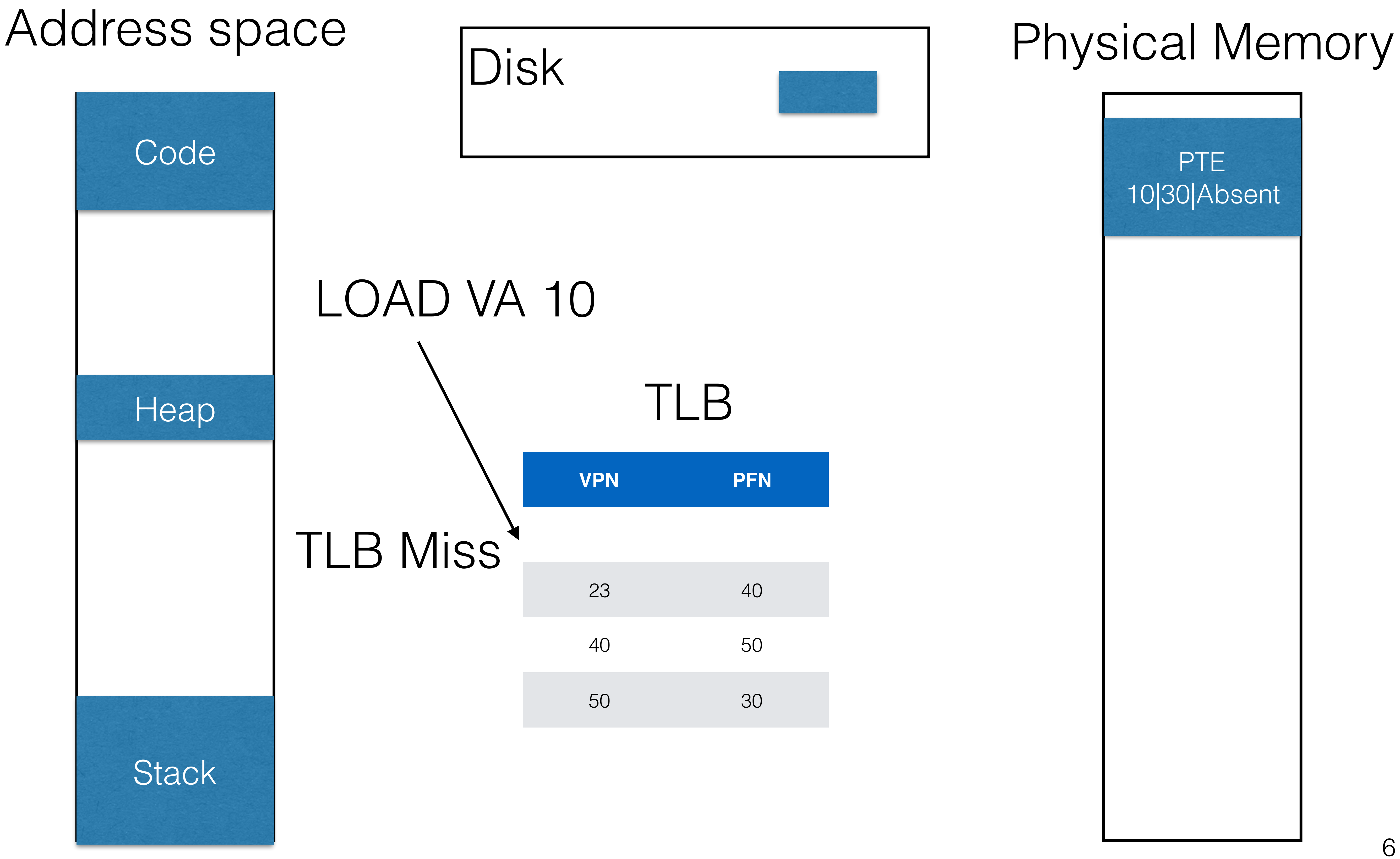
VPN	PFN
23	40
40	50
50	30

Swapping in



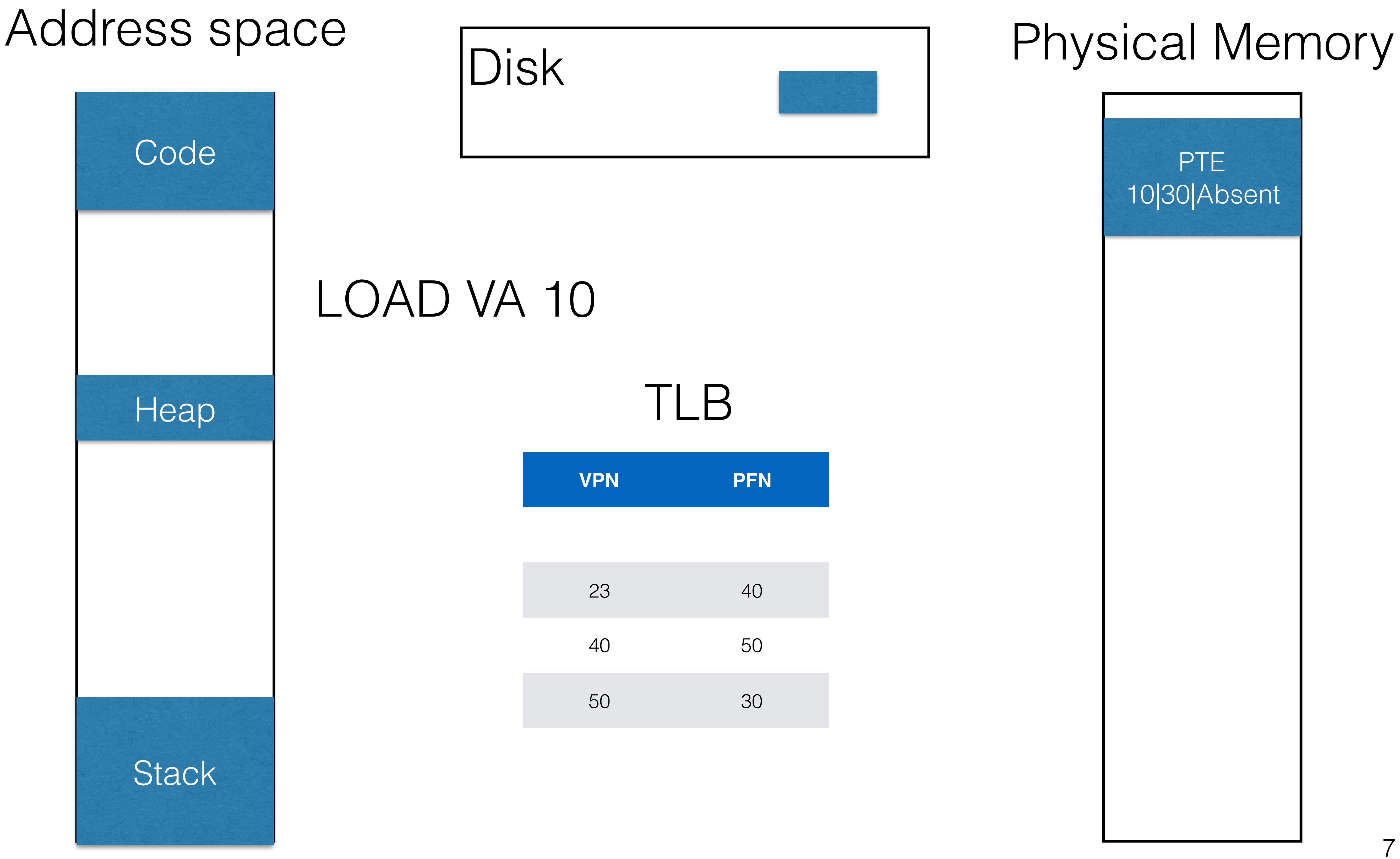
VPN	PFN
23	40
40	50
50	30

Swapping in



VPN	PFN
23	40
40	50
50	30

Swapping in

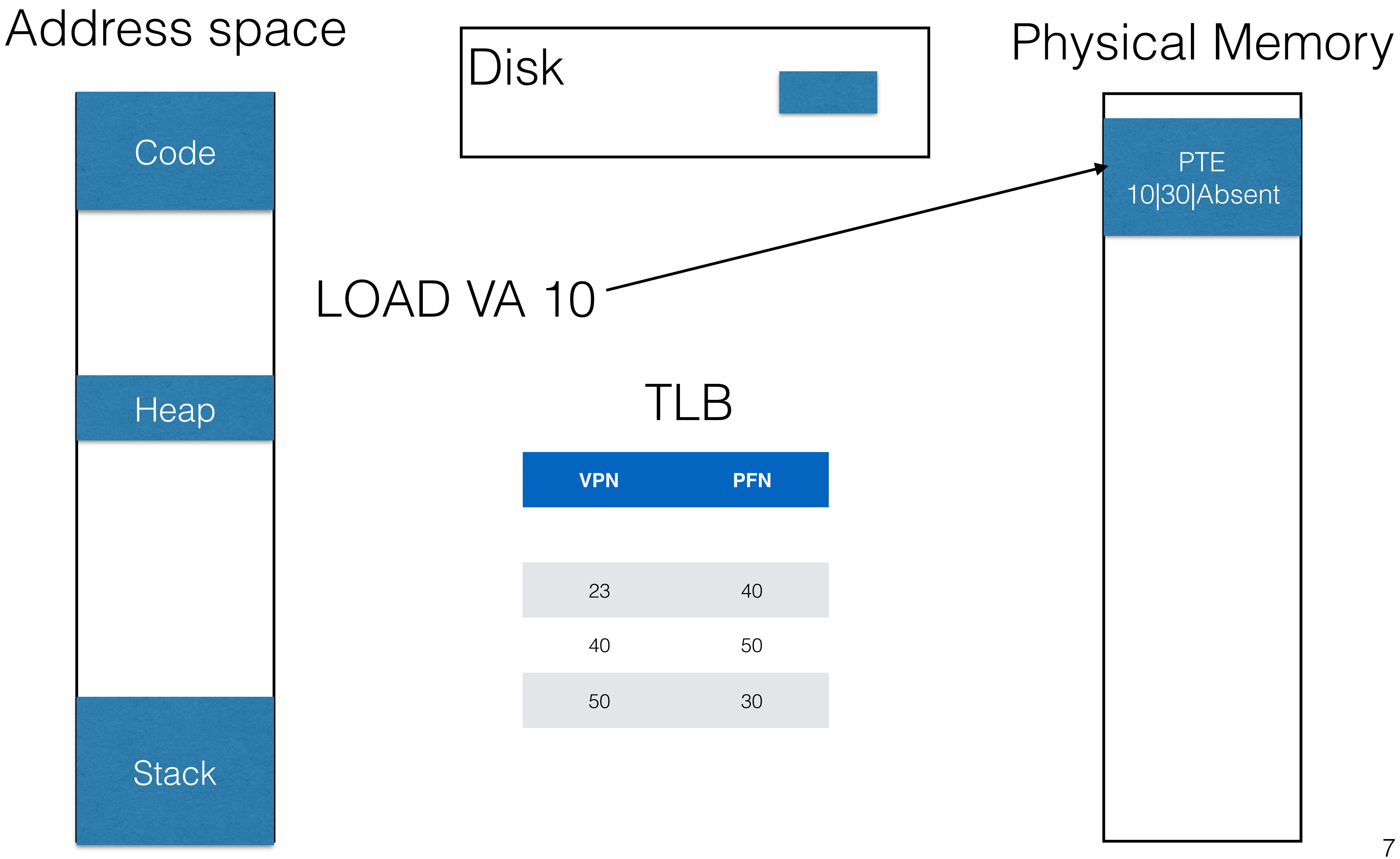


LOAD VA 10

TLB

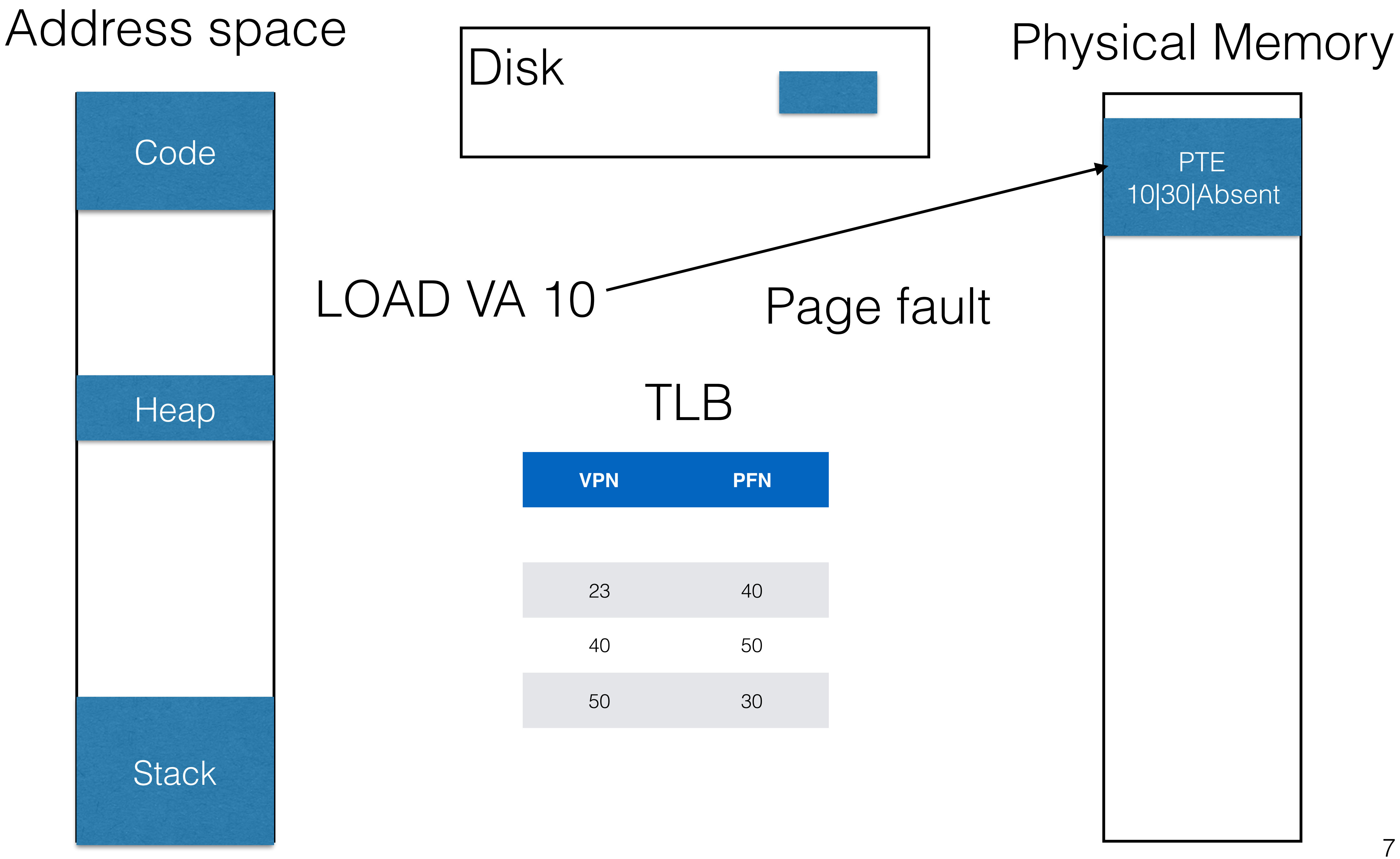
VPN	PFN
23	40
40	50
50	30

Swapping in



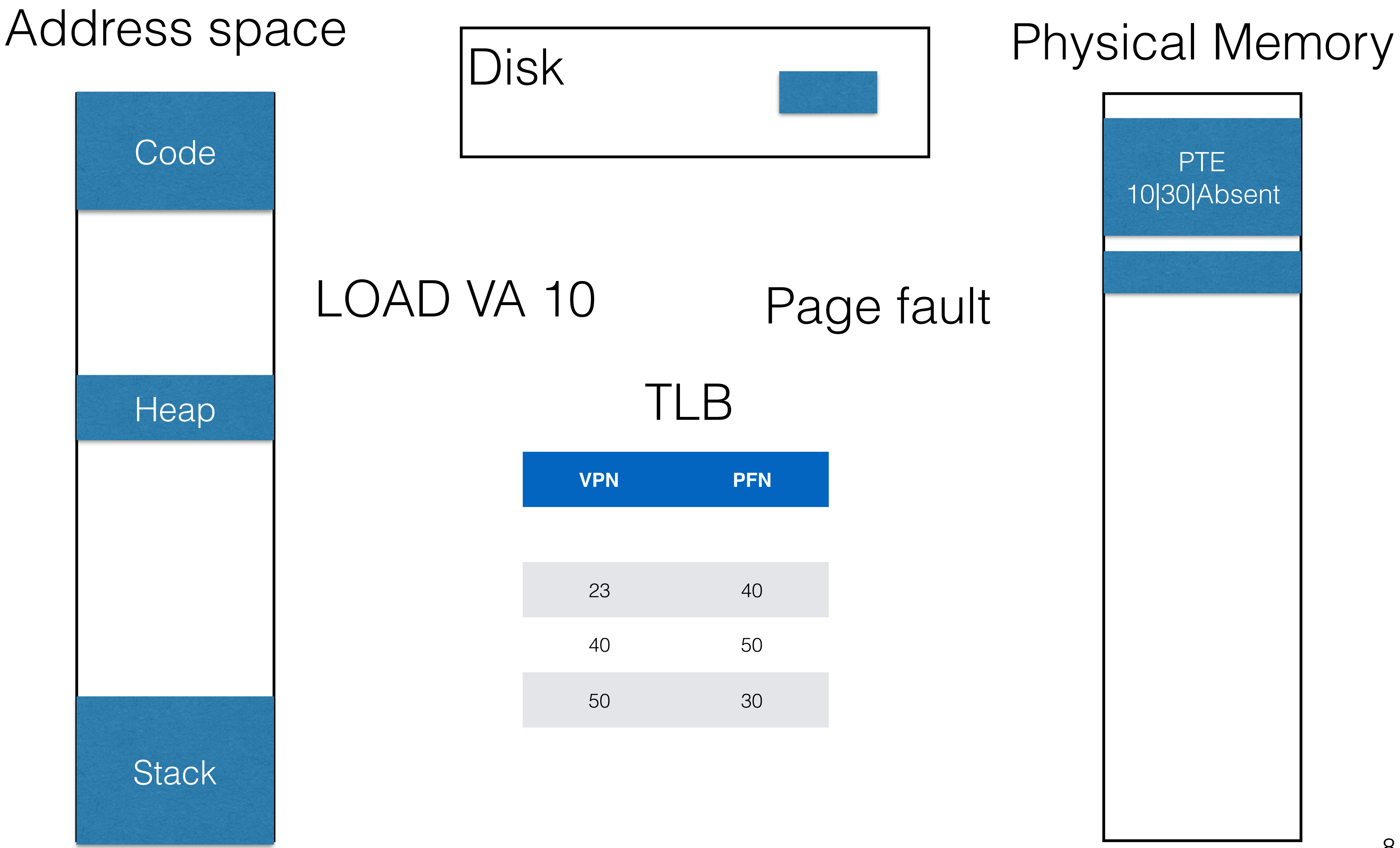
VPN	PFN
23	40
40	50
50	30

Swapping in



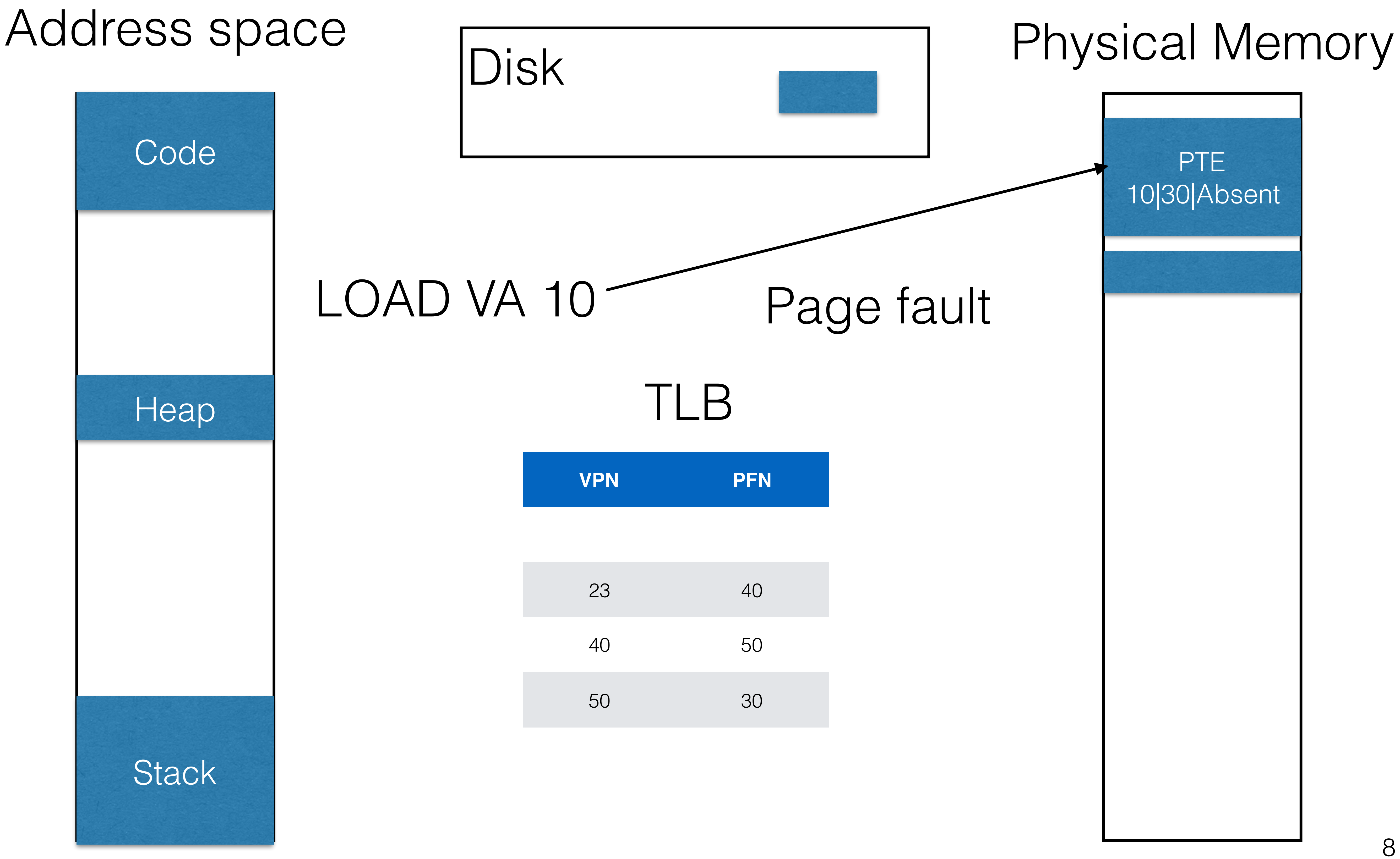
VPN	PFN
23	40
40	50
50	30

Swapping in



VPN	PFN
23	40
40	50
50	30

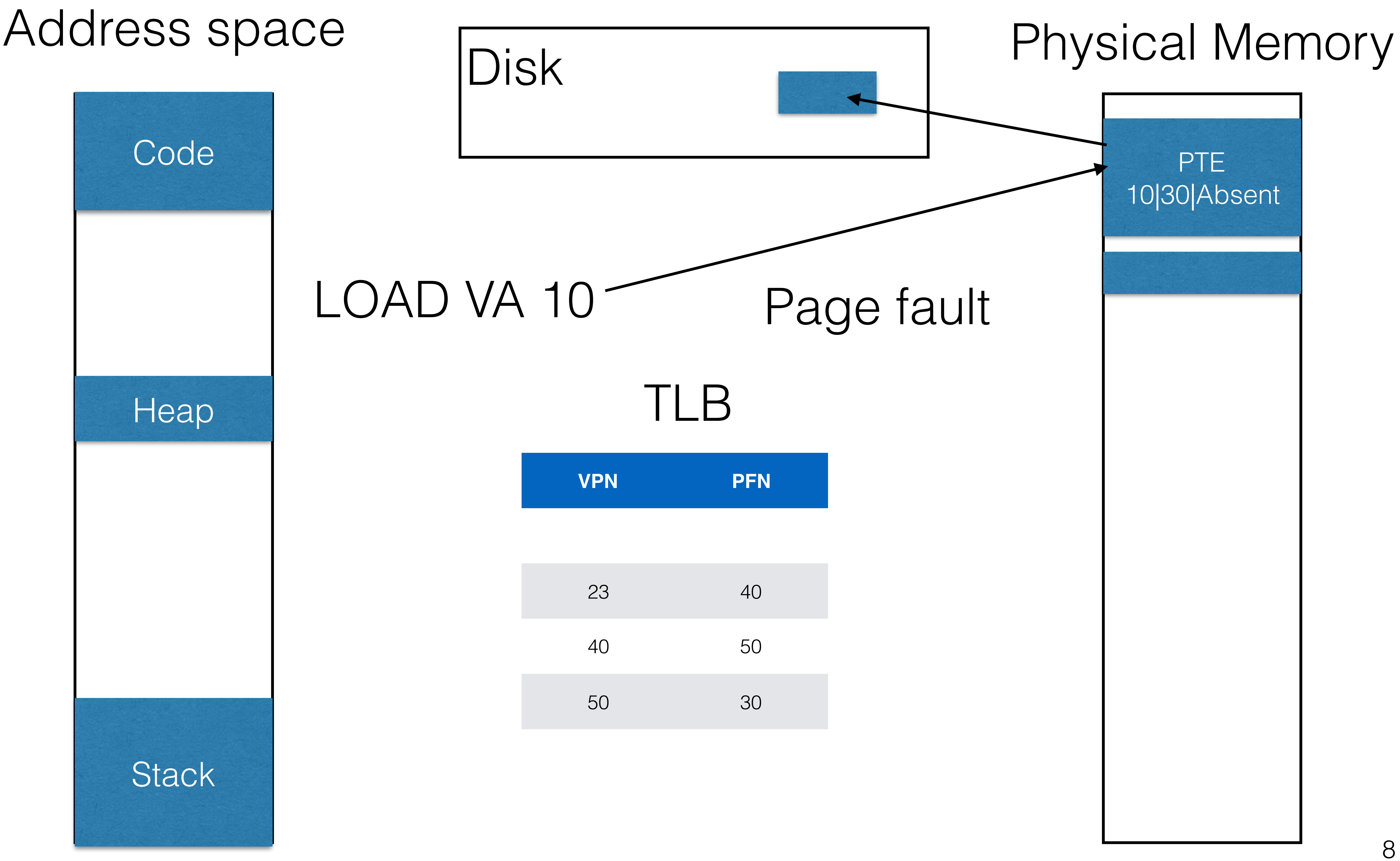
Swapping in



TLB

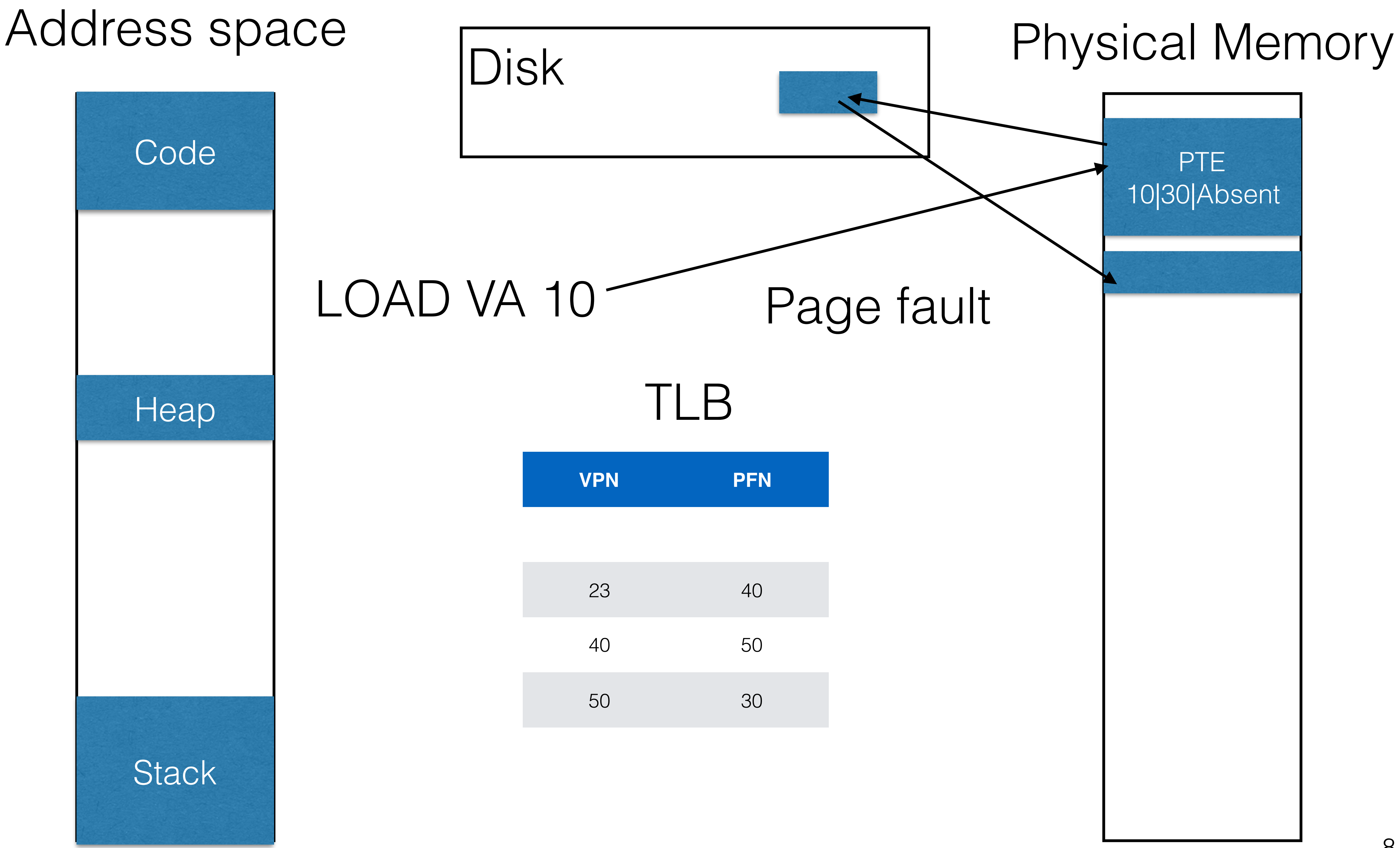
VPN	PFN
23	40
40	50
50	30

Swapping in



VPN	PFN
23	40
40	50
50	30

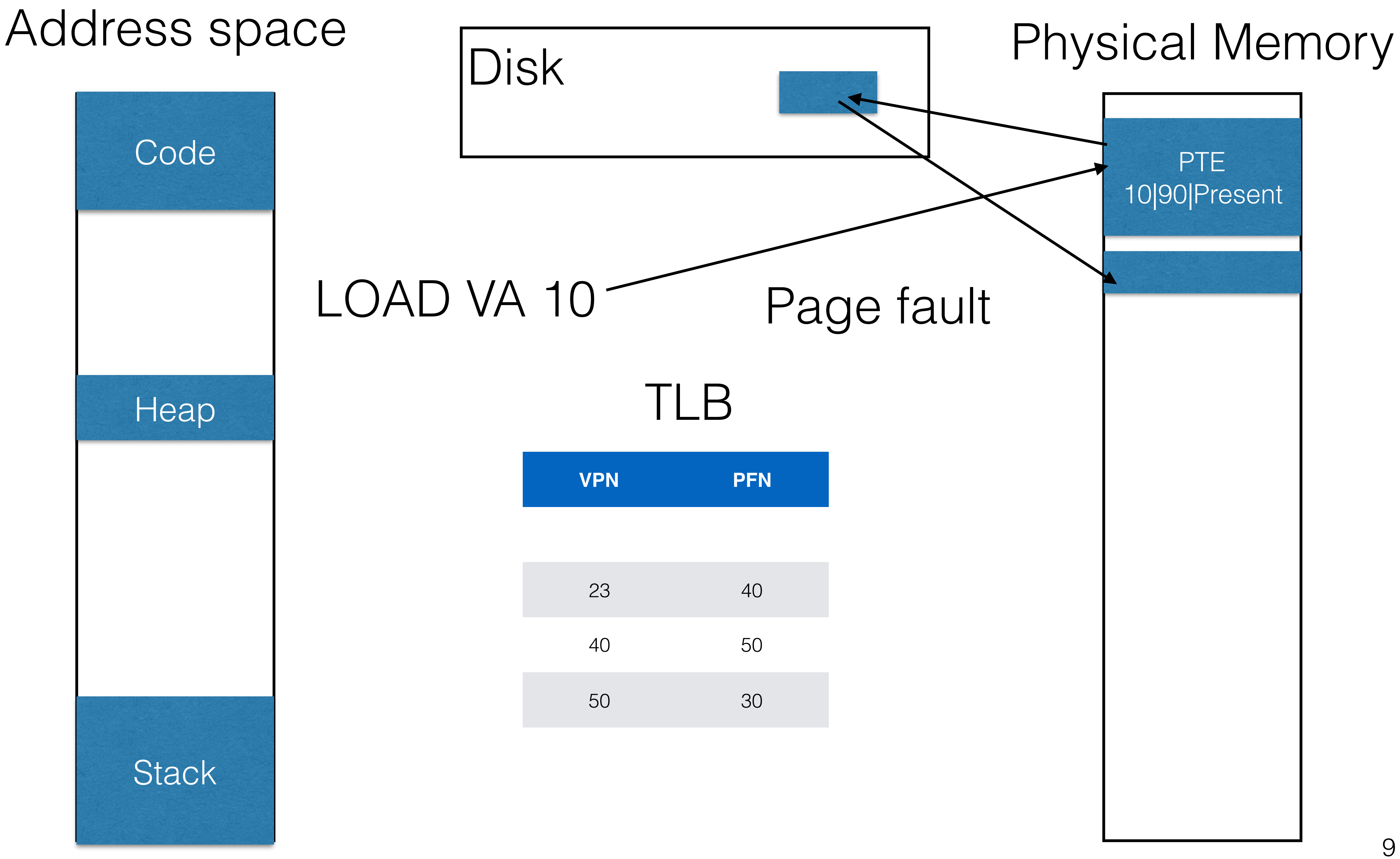
Swapping in



TLB

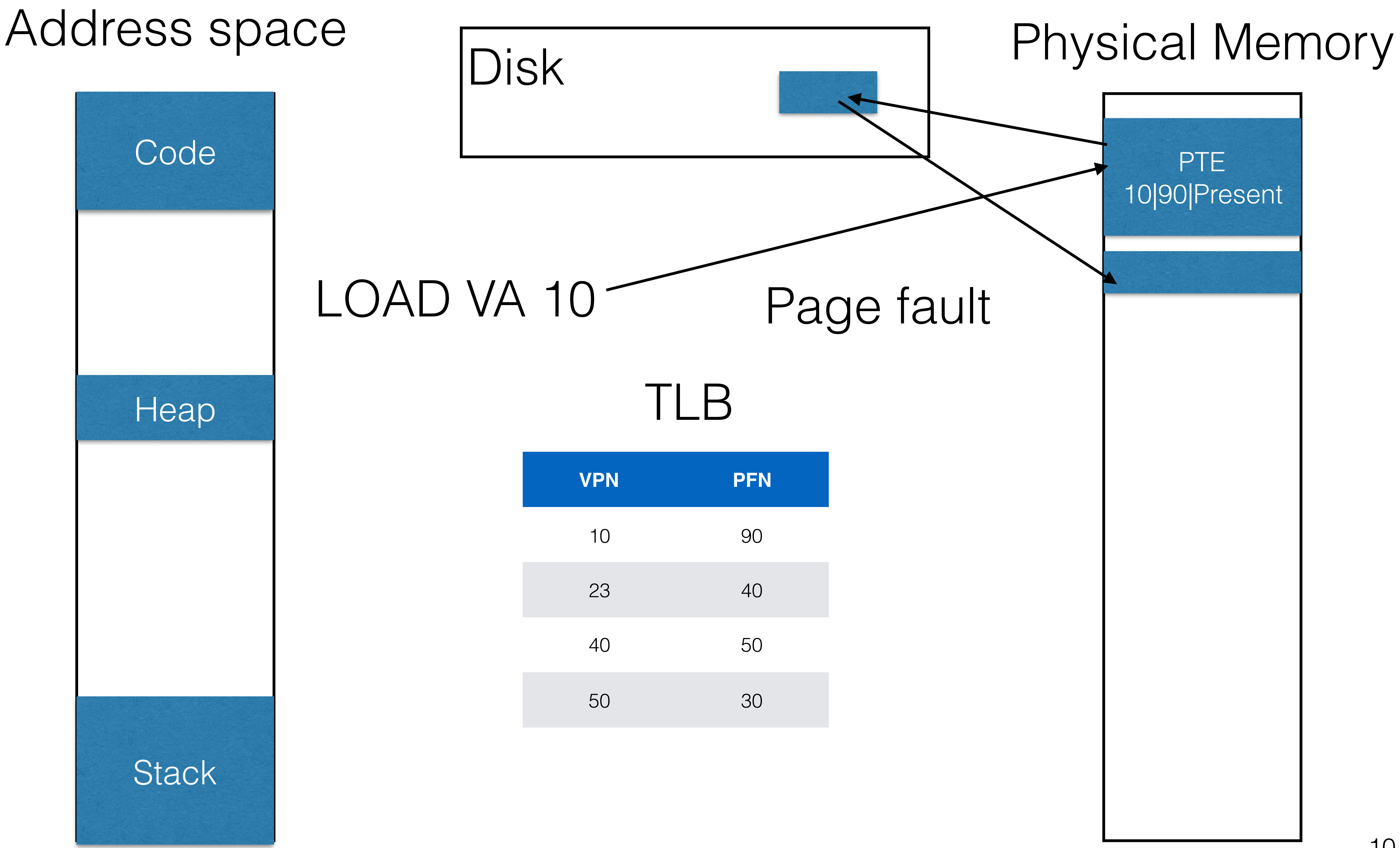
VPN	PFN
23	40
40	50
50	30

Swapping in



VPN	PFN
23	40
40	50
50	30

Swapping in



TLB

VPN	PFN
10	90
23	40
40	50
50	30

Page Replacement Policies - Optimal Replacement

Workload (page): 1,2,3,4,1,2,3,4,3,2,1
Cache size/Physical memory size: 3 pages

Page Replacement Policies - Optimal Replacement

Workload (page): 1,2,3,4,1,2,3,4,3,2,1
Cache size/Physical memory size: 3 pages

Optimal strategy: evict pages to be accessed furthest in future
—> Fewest possible cache misses

Page Replacement Policies - FIFO

Workload (page): 1,2,3,4,1,2,3,4,3,2,1
Cache size/Physical memory size: 4 pages

Page Replacement Policies - FIFO

Workload (page): 1,2,3,4,1,2,3,4,3,2,1
Cache size/Physical memory size: 4 pages

Strategy: evict pages based on FIFO

Page Replacement Policies -FIFO

Workload (page): 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
Cache size/Physical memory size: 3 pages/4 pages

Cache size: 3

Cache size: 4

Page Replacement Policies -FIFO

Workload (page): 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
Cache size/Physical memory size: 3 pages/4 pages

Cache size: 3

Acces	Hit	State
1	Miss	1
2	Miss	2, 1
3	Miss	3, 2, 1
4	Miss	4, 3, 2
1	Miss	1, 4, 3
2	Miss	2, 1, 4
5	Miss	5, 2, 1
1	**Hit**	5, 2, 1
2	**Hit**	5, 2, 1
3	Miss	3, 5, 2
4	Miss	4, 3, 5
5	**Hit**	4, 3, 5

Cache size: 4

Page Replacement Policies -FIFO

Workload (page): 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
Cache size/Physical memory size: 3 pages/4 pages

Cache size: 3

Acces	Hit	State
1	Miss	1
2	Miss	2, 1
3	Miss	3, 2, 1
4	Miss	4, 3, 2
1	Miss	1, 4, 3
2	Miss	2, 1, 4
5	Miss	5, 2, 1
1	**Hit**	5, 2, 1
2	**Hit**	5, 2, 1
3	Miss	3, 5, 2
4	Miss	4, 3, 5
5	**Hit**	4, 3, 5

Cache size: 4

Acces	Hit	State
1	Miss	1
2	Miss	2, 1
3	Miss	3, 2, 1
4	Miss	4, 3, 2, 1
1	**Hit**	4, 3, 2, 1
2	**Hit**	4, 3, 2, 1
5	Miss	5, 4, 3, 2
1	Miss	1, 5, 4, 2
2	Miss	2, 1, 5, 4
3	Miss	3, 2, 1, 5
4	Miss	4, 3, 2, 1
5	Miss	5, 4, 3, 2

Page Replacement Policies -FIFO
Belady's anomaly

Workload (page): 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
Cache size/Physical memory size: 3 pages/4 pages

Cache size: 3

Acces	Hit	State
1	Miss	1
2	Miss	2, 1
3	Miss	3, 2, 1
4	Miss	4, 3, 2
1	Miss	1, 4, 3
2	Miss	2, 1, 4
5	Miss	5, 2, 1
1	**Hit**	5, 2, 1
2	**Hit**	5, 2, 1
3	Miss	3, 5, 2
4	Miss	4, 3, 5
5	**Hit**	4, 3, 5

Cache size: 4

Acces	Hit	State
1	Miss	1
2	Miss	2, 1
3	Miss	3, 2, 1
4	Miss	4, 3, 2, 1
1	**Hit**	4, 3, 2, 1
2	**Hit**	4, 3, 2, 1
5	Miss	5, 4, 3, 2
1	Miss	1, 5, 4, 2
2	Miss	2, 1, 5, 4
3	Miss	3, 2, 1, 5
4	Miss	4, 3, 2, 1
5	Miss	5, 4, 3, 2

Page Replacement Policies - Random

Workload (page): 1,2,3,4,1,2,3,4,3,2,1
Cache size/Physical memory size: 4 pages

Page Replacement Policies - Random

Workload (page): 1,2,3,4,1,2,3,4,3,2,1
Cache size/Physical memory size: 4 pages

Random strategy: randomly evict pages

Page Replacement Policies -History based (LRU and LFU)

Workload (page): 1,2,3,4,1,2,3,4,3,2,1
Cache size/Physical memory size: 3 pages

Page Replacement Policies -History based (LRU and LFU)

Workload (page): 1,2,3,4,1,2,3,4,3,2,1
Cache size/Physical memory size: 3 pages

Strategy:

1. Least frequently used - evict least frequently used page
2. Least recently used - evict least recently used page

LRU implementation

LRU implementation

- On each access, update time of page

LRU implementation

- On each access, update time of page
- When looking for eviction:

LRU implementation

- On each access, update time of page
- When looking for eviction:
  - Search for all candidate sets (millions of pages)

LRU implementation

- On each access, update time of page
- When looking for eviction:
    - Search for all candidate sets (millions of pages)
    - Find least recently used

LRU implementation

- On each access, update time of page
- When looking for eviction:
  - Search for all candidate sets (millions of pages)
  - Find least recently used
- Huge overhead!

LRU implementation (Appx - Clock Hand Algo)

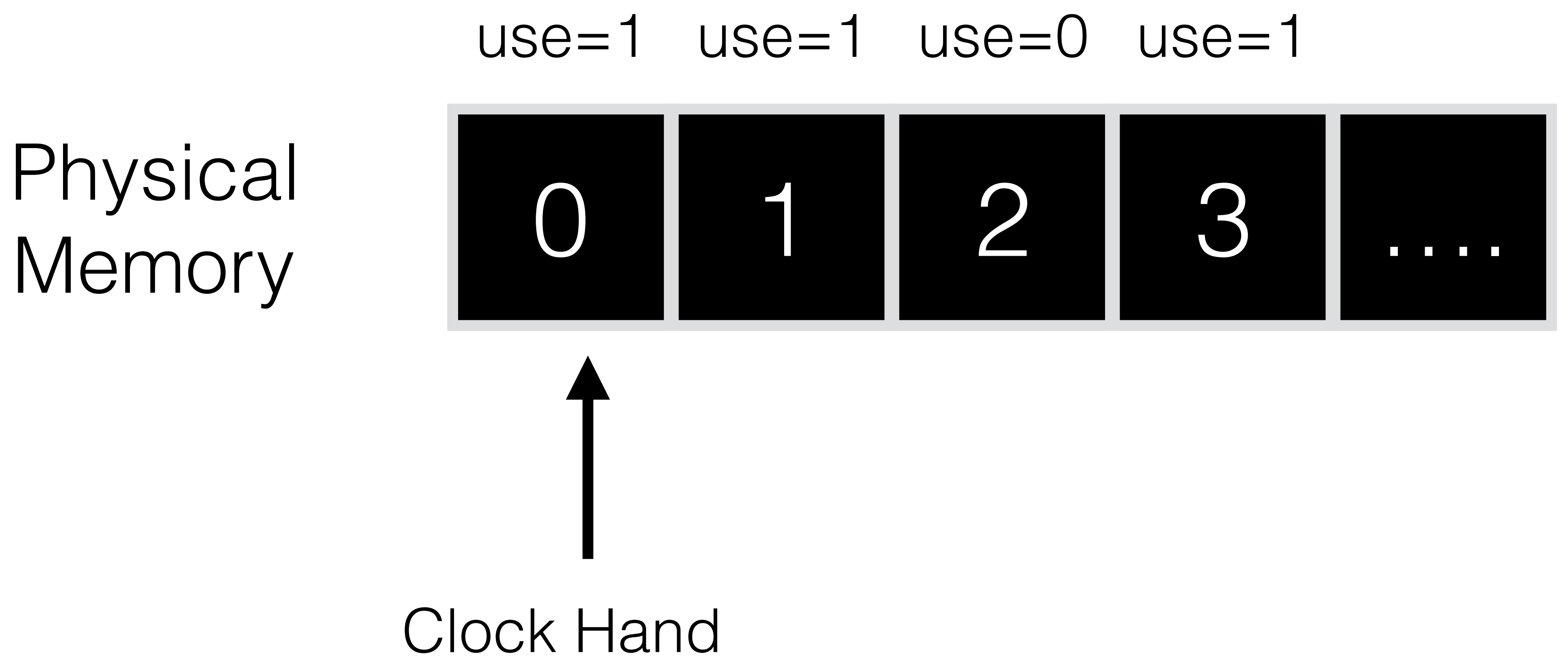
LRU implementation (Appx - Clock Hand Algo)

- On each access, set reference bit for page

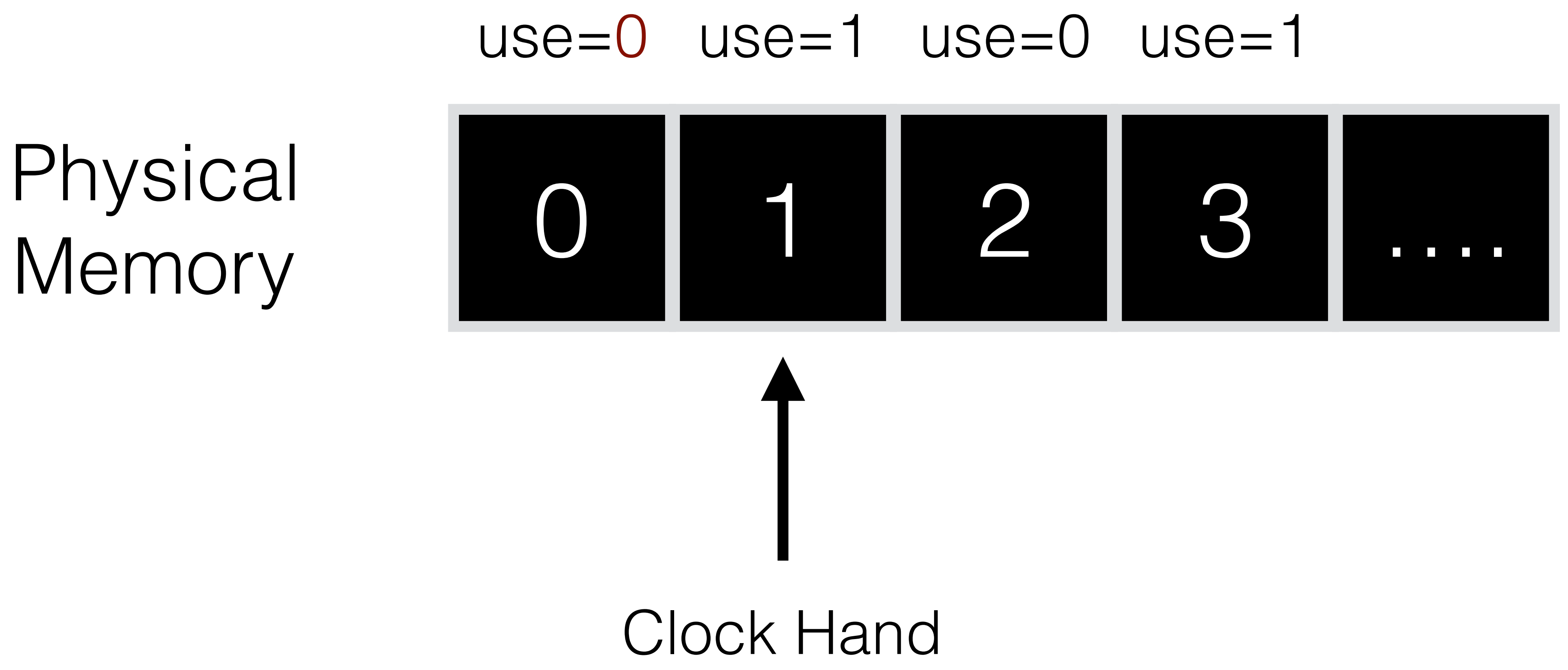
LRU implementation (Appx - Clock Hand Algo)

- On each access, set reference bit for page
- Clock algorithm - look for nearest page without set reference bit

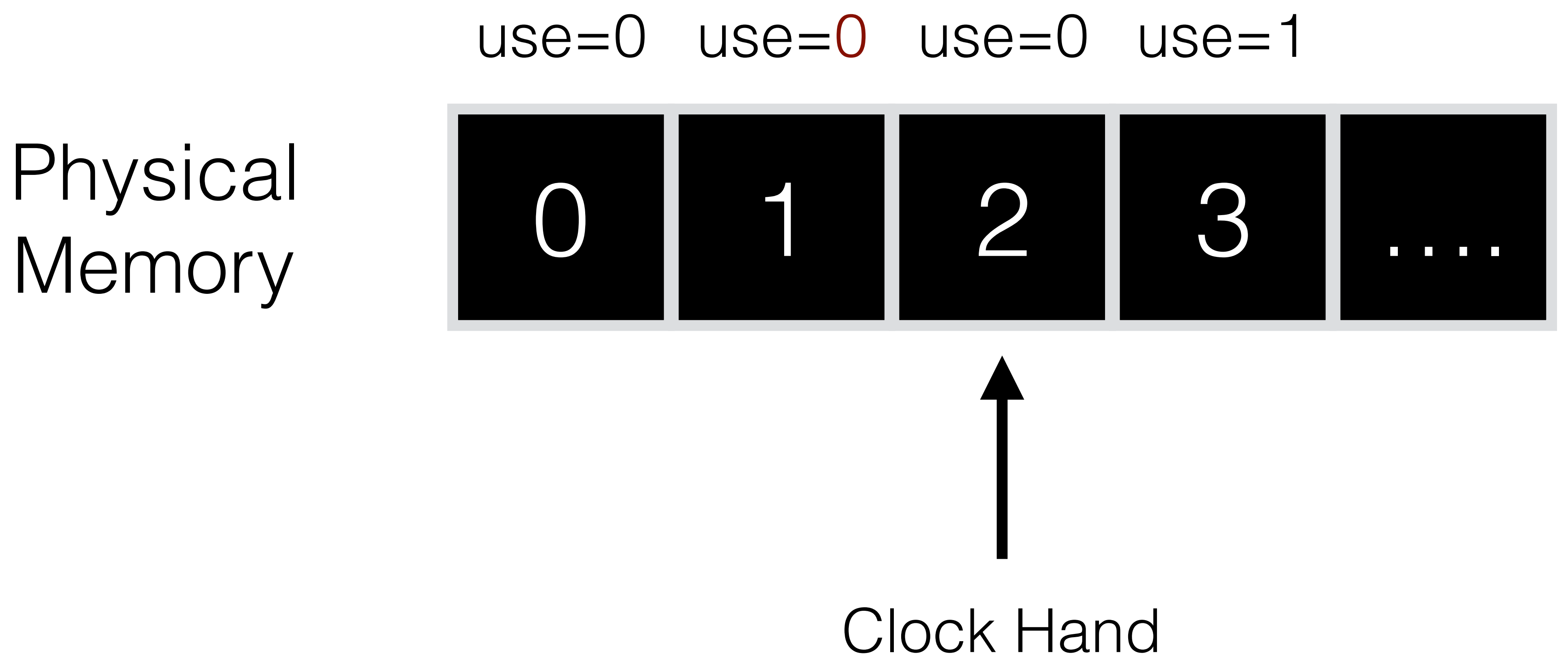
LRU implementation (Appx - Clock Hand Algo)



LRU implementation (Appx - Clock Hand Algo)

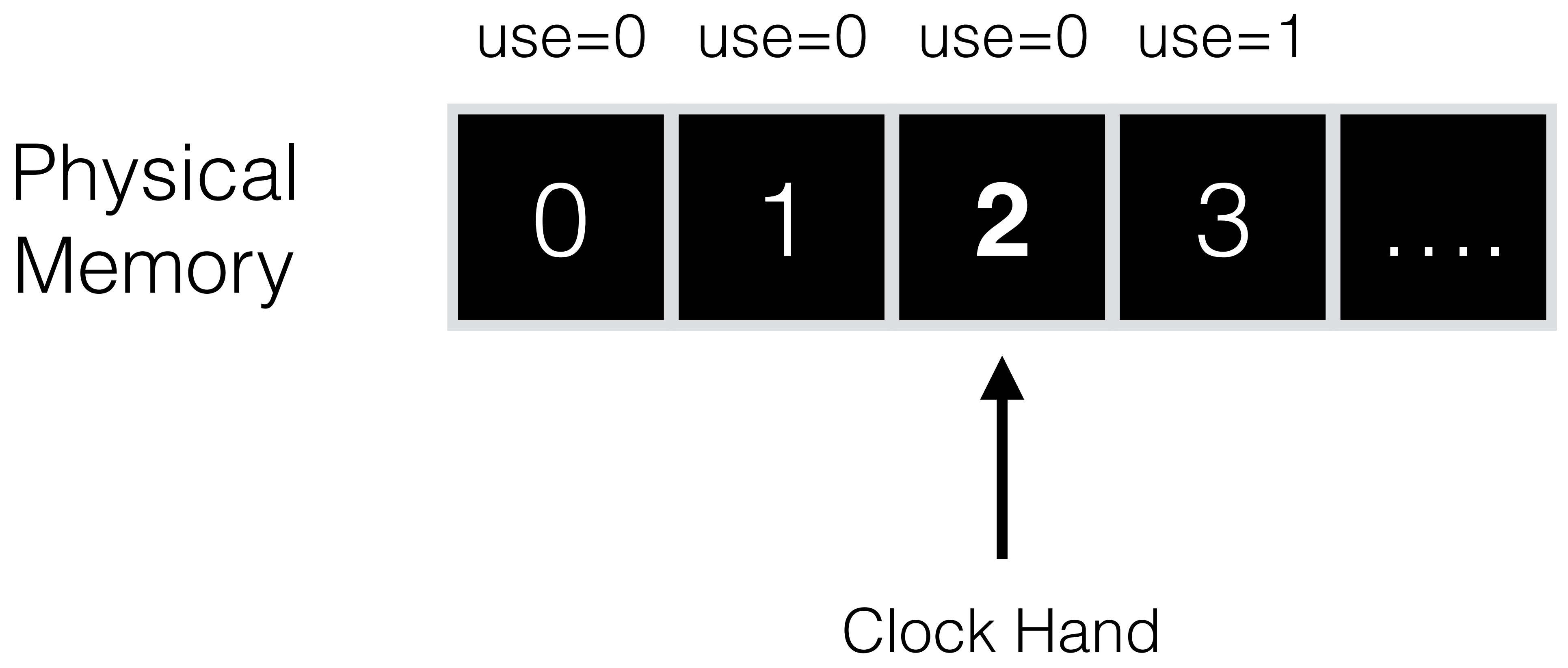


LRU implementation (Appx - Clock Hand Algo)



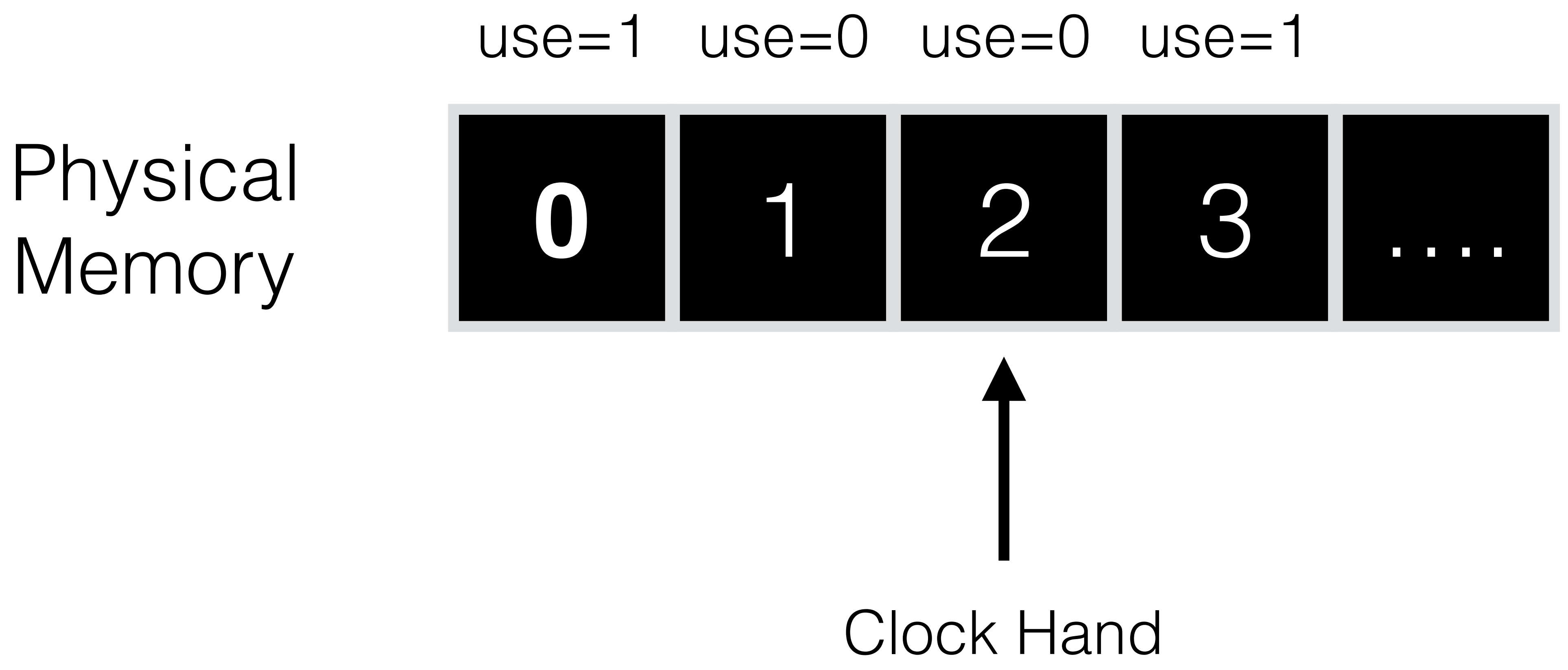
LRU implementation (Appx - Clock Hand Algo)

Evict Page 2: Not recently used

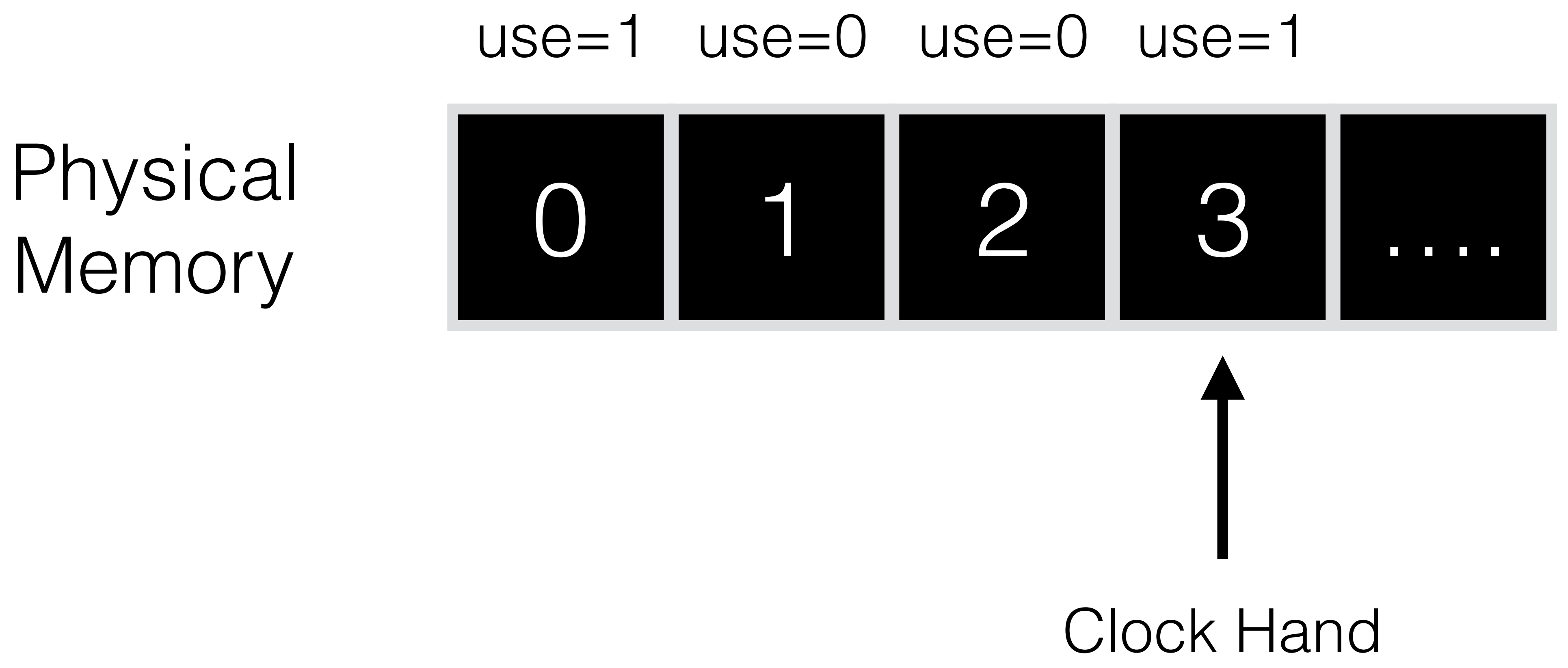


LRU implementation (Appx - Clock Hand Algo)

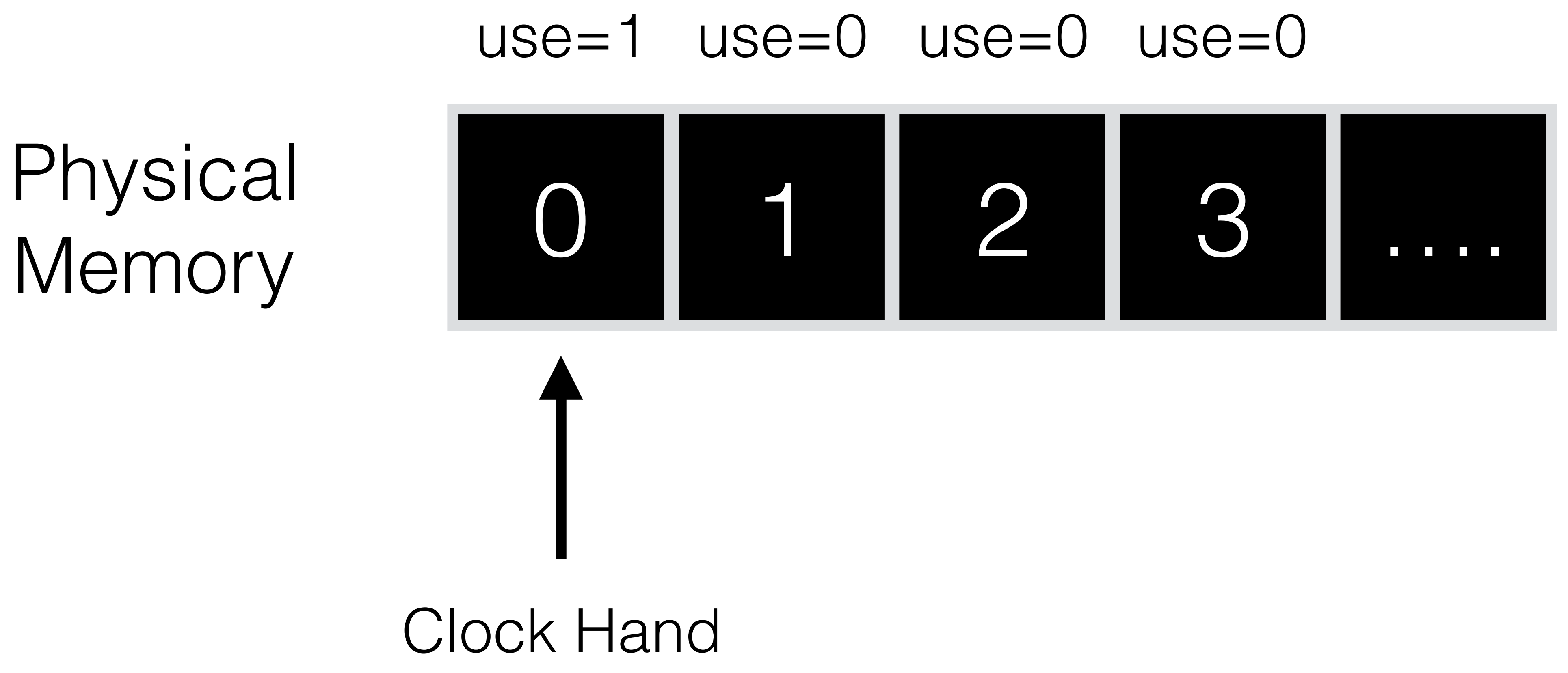
Page 0 is accessed



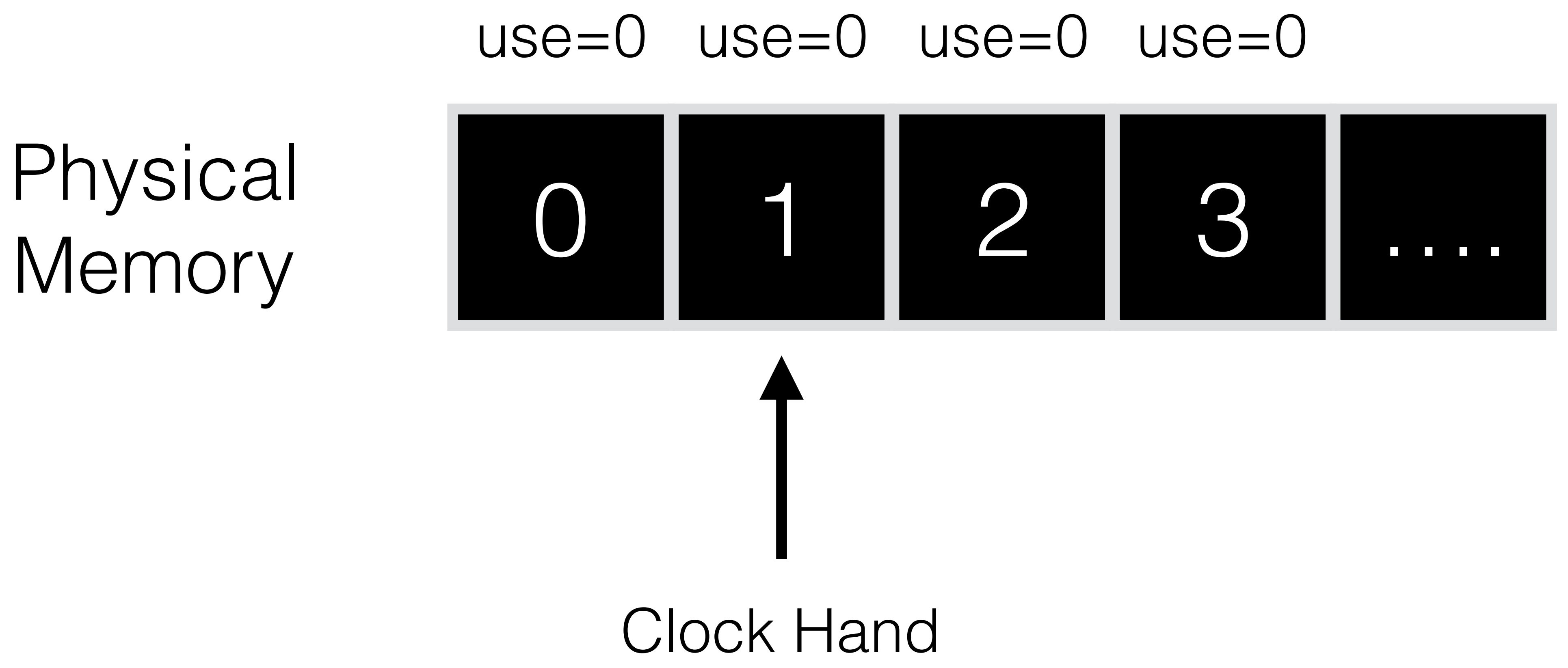
LRU implementation (Appx - Clock Hand Algo)



LRU implementation (Appx - Clock Hand Algo)

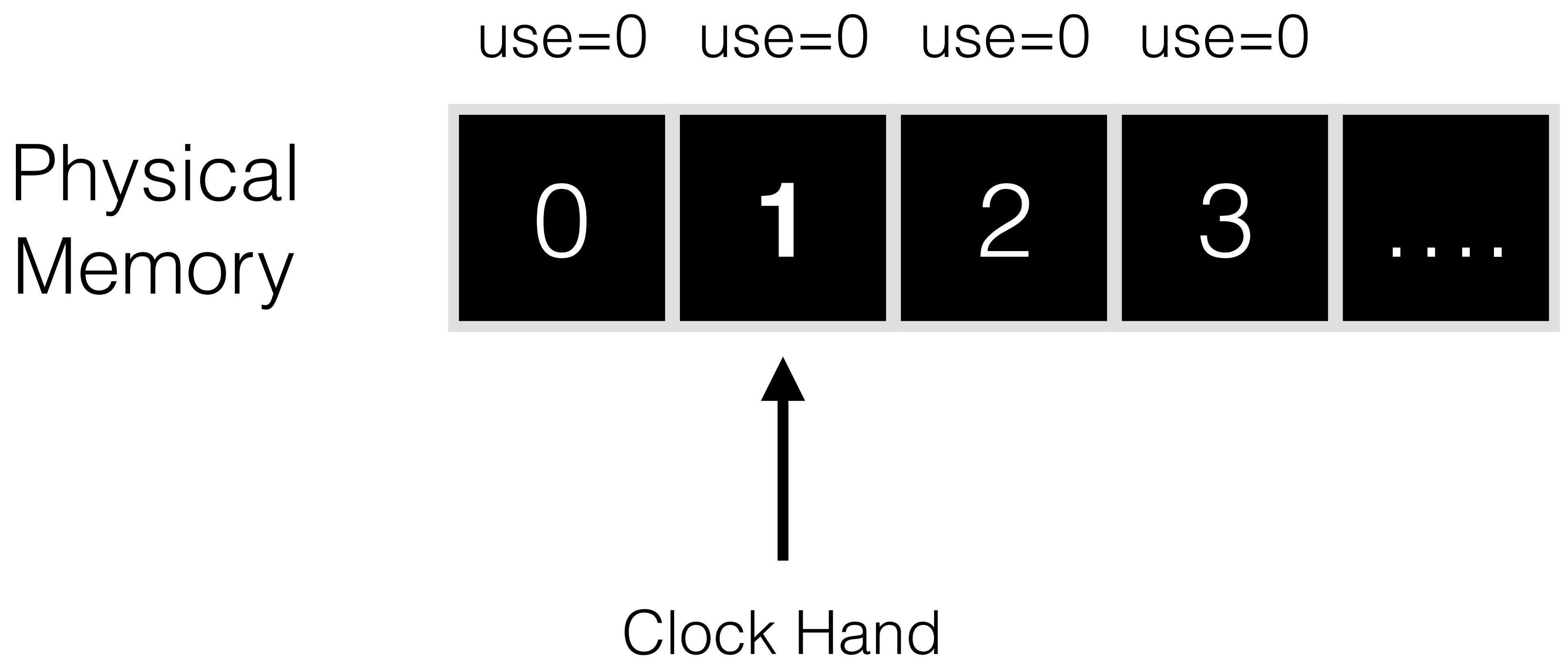


LRU implementation (Appx - Clock Hand Algo)



LRU implementation (Appx - Clock Hand Algo)

Evict Page 1: Not recently used



Other Factors

Other Factors

- Assume page is both on disk and RAM

Other Factors

- Assume page is both on disk and RAM
- Do we have to write the evicted page to disk?

Other Factors

- Assume page is both on disk and RAM
- Do we have to write the evicted page to disk?
    - If page is clean?

Other Factors

- Assume page is both on disk and RAM
- Do we have to write the evicted page to disk?
  - If page is clean?
    - NO!

Other Factors

- Assume page is both on disk and RAM
- Do we have to write the evicted page to disk?
  - If page is clean?
    - NO!
  - If page is dirty?

Other Factors

- Assume page is both on disk and RAM
- Do we have to write the evicted page to disk?
    - If page is clean?
        - NO!
    - If page is dirty?
        - Yes!

Other Factors

Other Factors

- When to swap in?

Other Factors

- When to swap in?
    - Demand paging: swap in when needed

Other Factors

- When to swap in?
  - Demand paging: swap in when needed
  - Prefetching: swap in a page ahead of demand (anticipating demand)

Other Factors

- When to swap in?
  - Demand paging: swap in when needed
  - Prefetching: swap in a page ahead of demand (anticipating demand)
    - When likely?

Other Factors

- When to swap in?
  - Demand paging: swap in when needed
  - Prefetching: swap in a page ahead of demand (anticipating demand)
    - When likely?
      - Code page P brought to memory, P+1 also likely

Other Factors

- When to swap in?
  - Demand paging: swap in when needed
  - Prefetching: swap in a page ahead of demand (anticipating demand)
    - When likely?
      - Code page P brought to memory, P+1 also likely
- When to write to disk

Other Factors

- When to swap in?
  - Demand paging: swap in when needed
  - Prefetching: swap in a page ahead of demand (anticipating demand)
    - When likely?
      - Code page P brought to memory, P+1 also likely
- When to write to disk
  - One at a time

Other Factors

- When to swap in?
  - Demand paging: swap in when needed
  - Prefetching: swap in a page ahead of demand (anticipating demand)
    - When likely?
      - Code page P brought to memory, P+1 also likely
- When to write to disk
  - One at a time
  - Clustered writes - preferred - 1 large write quicker than multiple smaller writes

Free space management

Free space management

- Advantage of paging:

Free space management

- Advantage of paging:
  - Fixed size units. Easier to maintain free space.

Free space management

- Advantage of paging:
  - Fixed size units. Easier to maintain free space.
- Non-fixed size units used where?

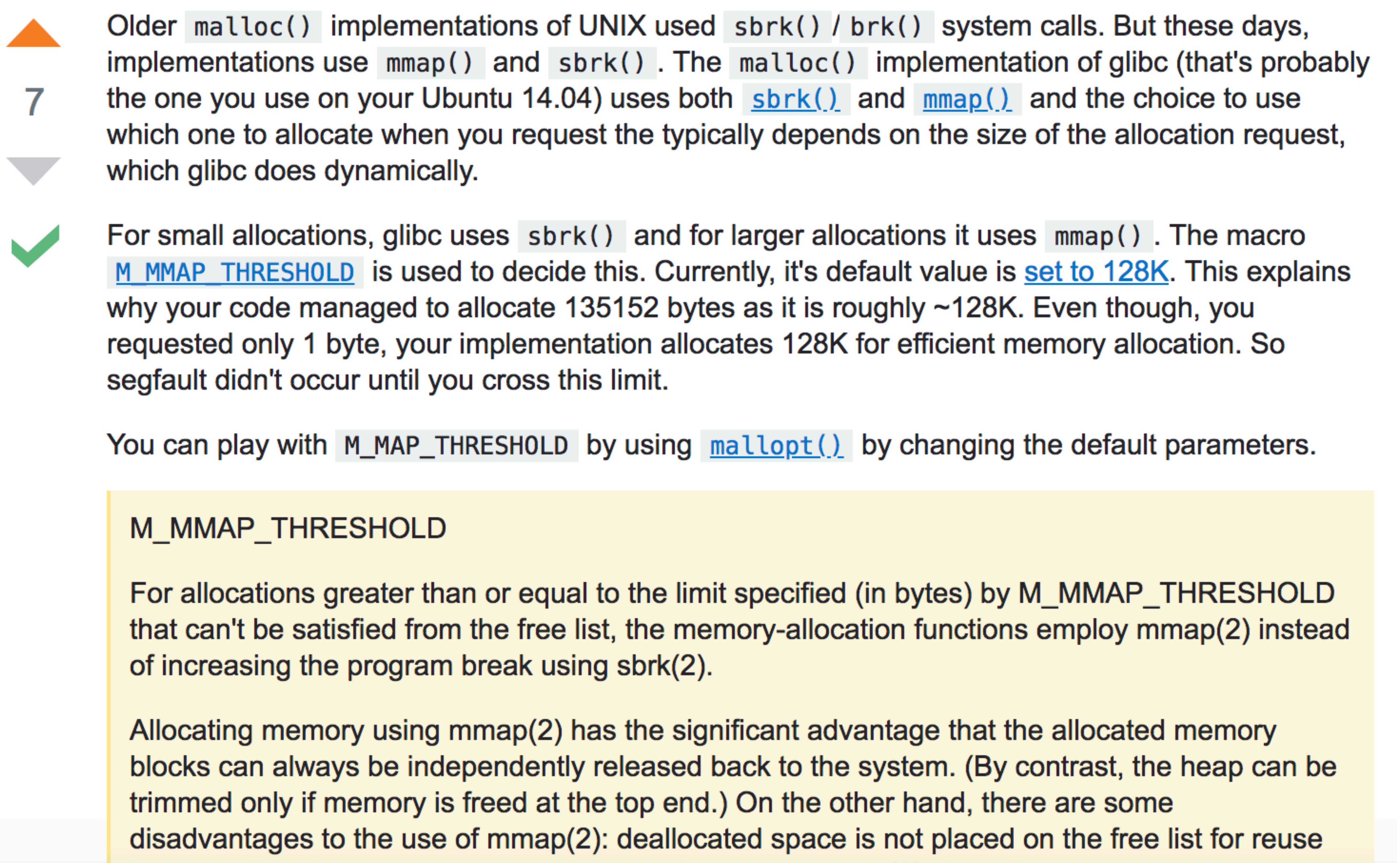
Free space management

- Advantage of paging:
  - Fixed size units. Easier to maintain free space.
- Non-fixed size units used where?
  - Segmentation

Free space management

- Advantage of paging:
  - Fixed size units. Easier to maintain free space.
- Non-fixed size units used where?
  - Segmentation
  - Malloc?

Malloc Caveat



7

Older `malloc()` implementations of UNIX used `sbrk()` / `brk()` system calls. But these days, implementations use `mmap()` and `sbrk()`. The `malloc()` implementation of glibc (that's probably the one you use on your Ubuntu 14.04) uses both `sbrk()` and `mmap()` and the choice to use which one to allocate when you request the typically depends on the size of the allocation request, which glibc does dynamically.

For small allocations, glibc uses `sbrk()` and for larger allocations it uses `mmap()`. The macro `M_MMAP_THRESHOLD` is used to decide this. Currently, it's default value is set to 128K. This explains why your code managed to allocate 135152 bytes as it is roughly ~128K. Even though, you requested only 1 byte, your implementation allocates 128K for efficient memory allocation. So segfault didn't occur until you cross this limit.

You can play with `M_MAP_THRESHOLD` by using `mallopt()` by changing the default parameters.

> M_MMAP_THRESHOLD
>
> For allocations greater than or equal to the limit specified (in bytes) by M_MMAP_THRESHOLD that can't be satisfied from the free list, the memory-allocation functions employ mmap(2) instead of increasing the program break using sbrk(2).
>
> Allocating memory using mmap(2) has the significant advantage that the allocated memory blocks can always be independently released back to the system. (By contrast, the heap can be trimmed only if memory is freed at the top end.) On the other hand, there are some disadvantages to the use of mmap(2): deallocated space is not placed on the free list for reuse

Malloc & Free

Malloc & Free

Malloc & Free

- Interface of malloc and free

Malloc & Free

- Interface of malloc and free
  - Malloc takes size as argument —> returns pointer

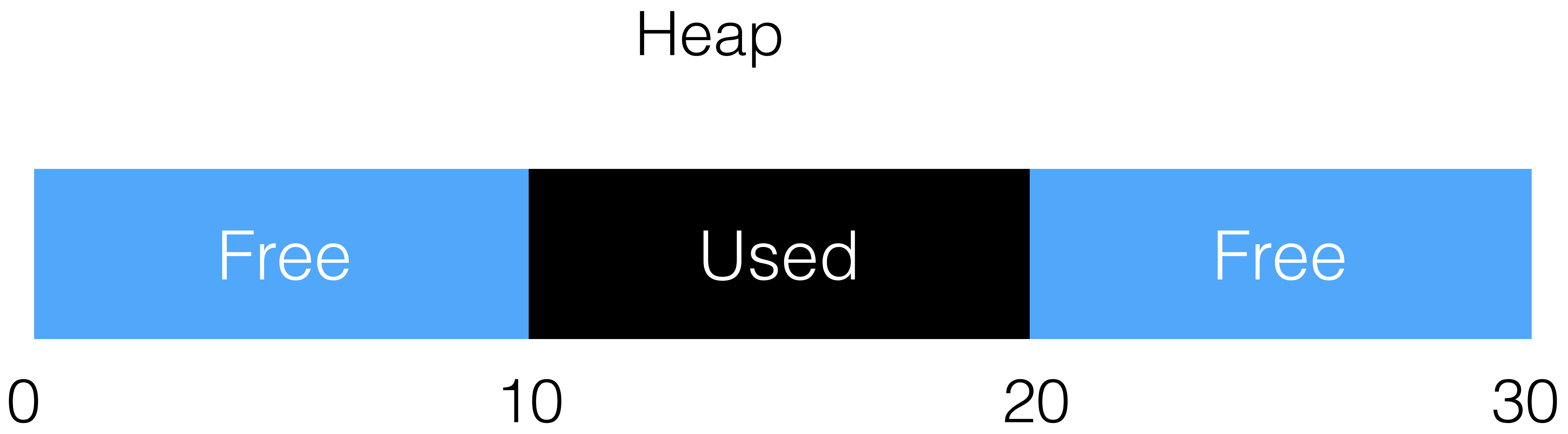
Malloc & Free

- Interface of malloc and free
  - Malloc takes size as argument —> returns pointer
  - Free takes a pointer and frees the corresponding chunk

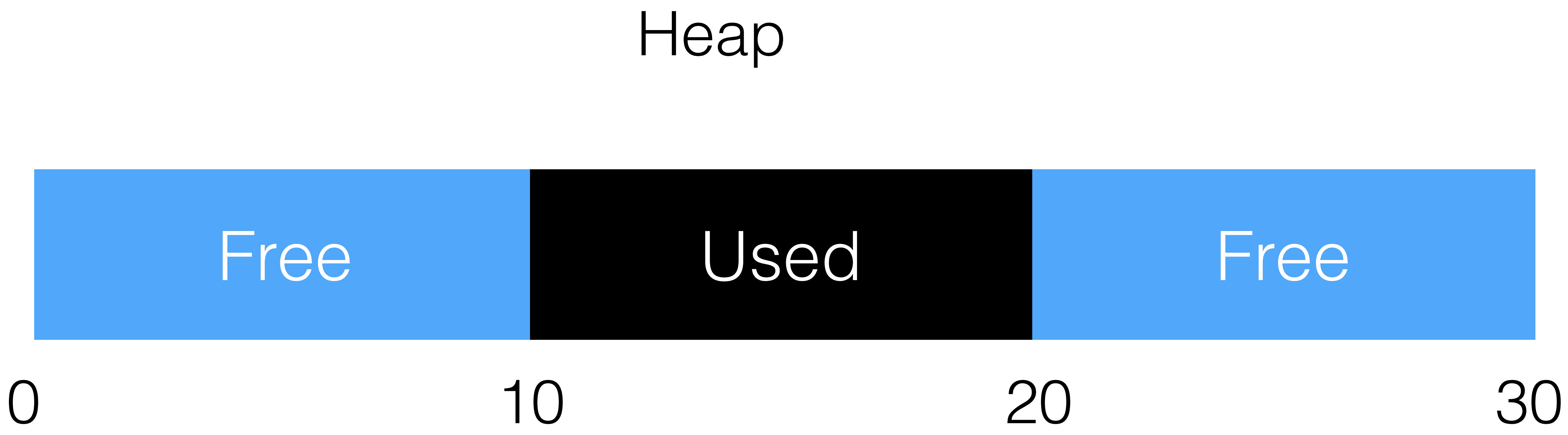
Malloc & Free

- Interface of malloc and free
  - Malloc takes size as argument —> returns pointer
  - Free takes a pointer and frees the corresponding chunk
    - Does not provide the size! How does it know the size?

Revisiting External Fragmentation

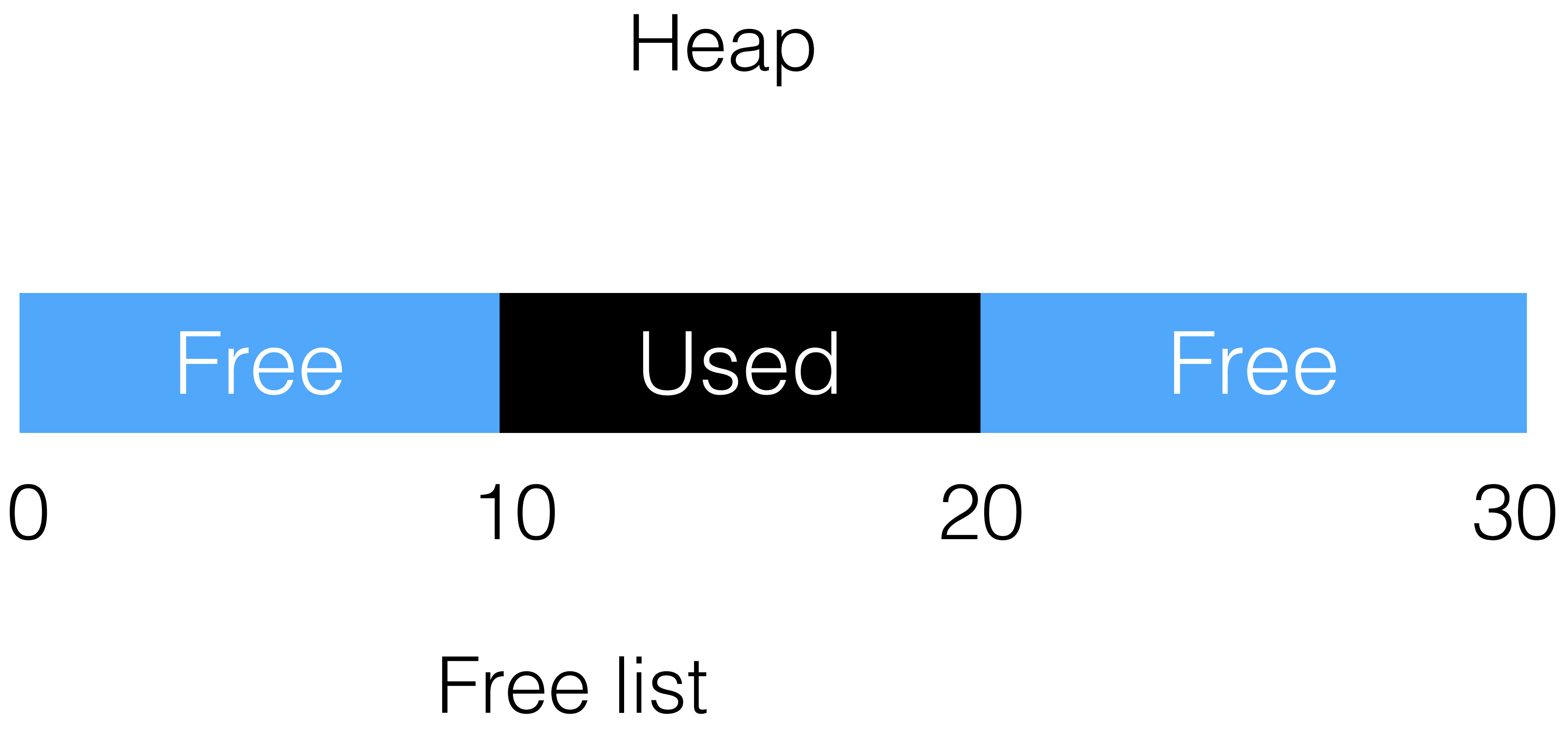


Revisiting External Fragmentation

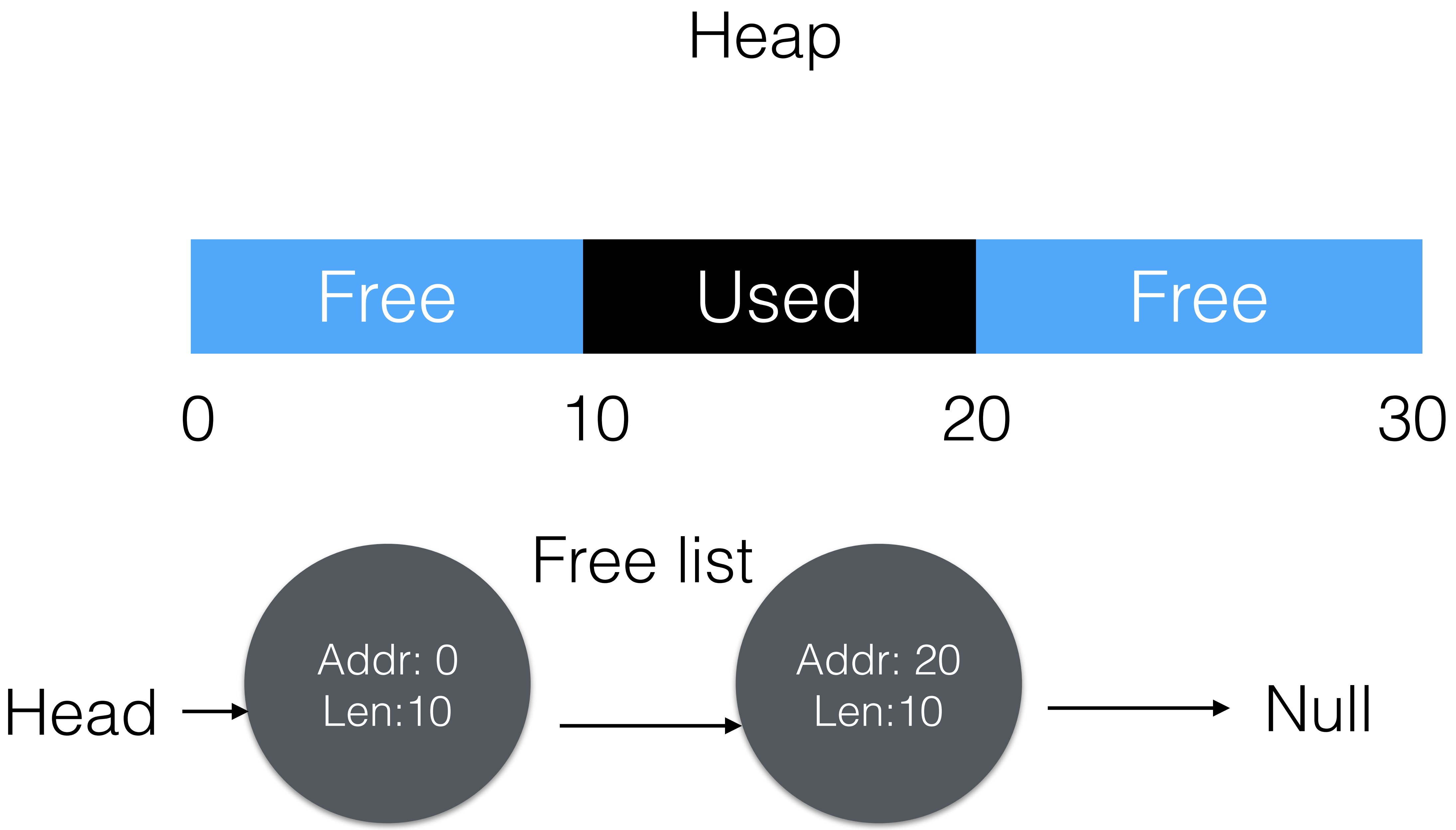


Request for 15 bytes will fail

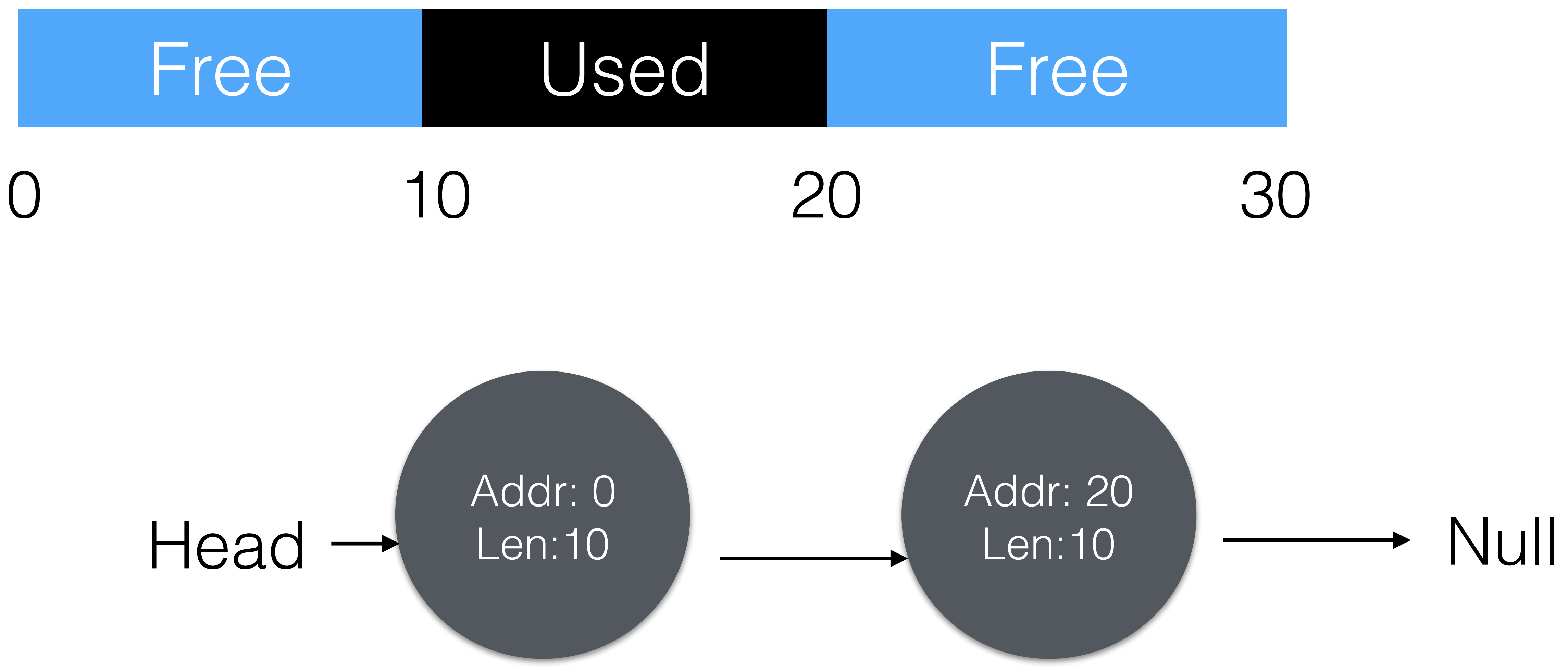
Free list



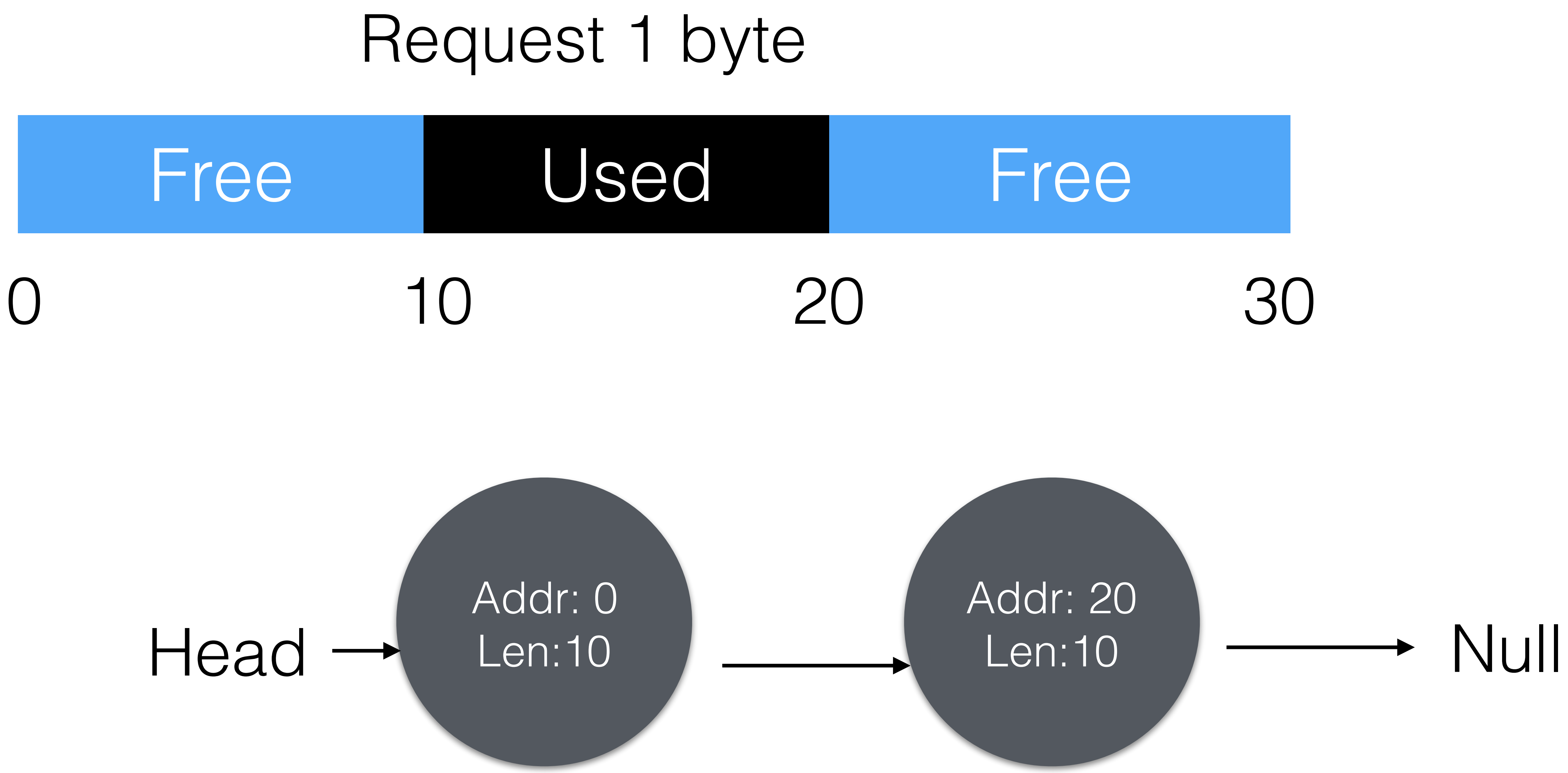
Free list


Heap
Free
Used
Free
0
10
20
30
Free list
Head
Addr: 0
Len:10
Addr: 20
Len:10
Null

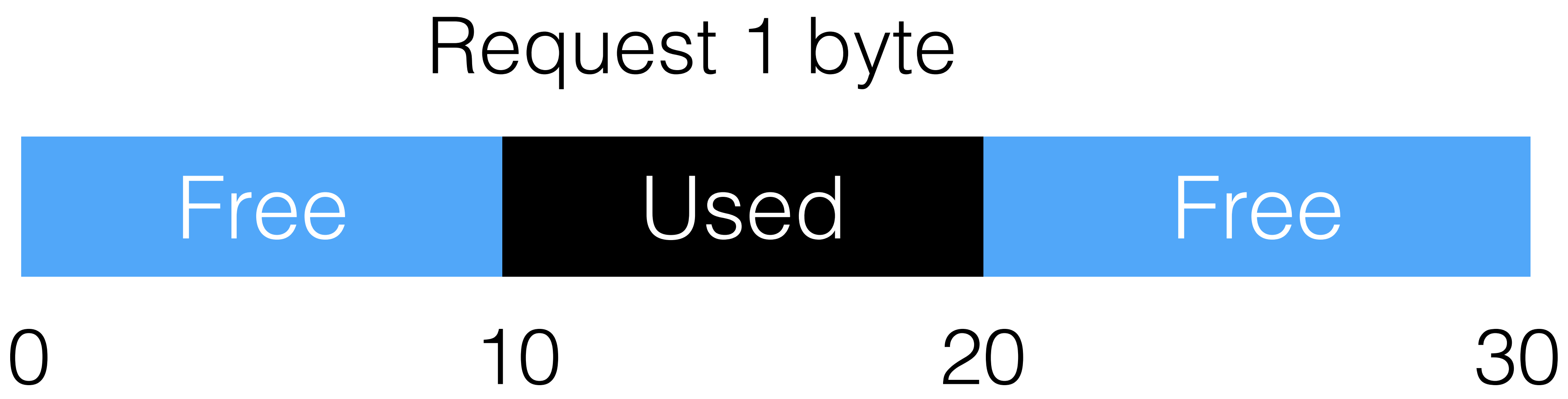
Split

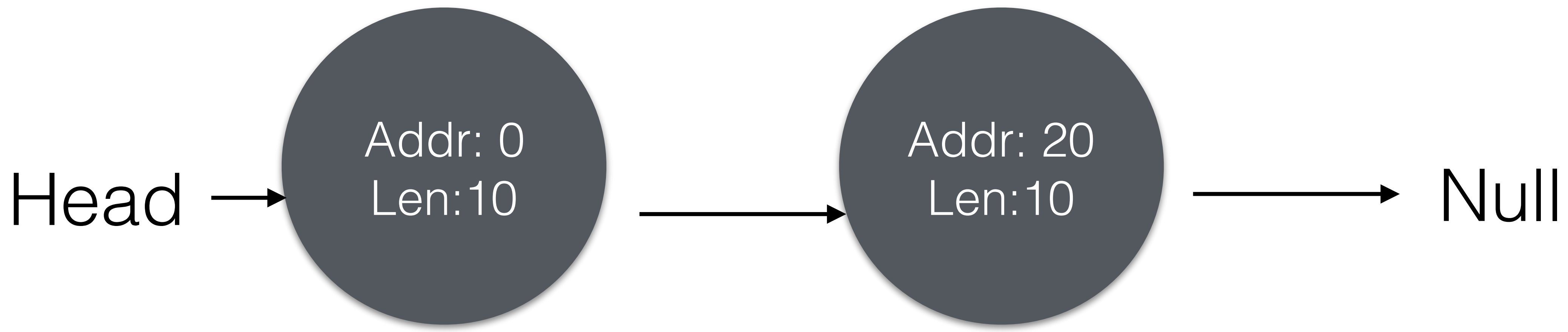


Split


Request 1 byte
Free
Used
Free
0
10
20
30
Head
Addr: 0
Len:10
Addr: 20
Len:10
Null

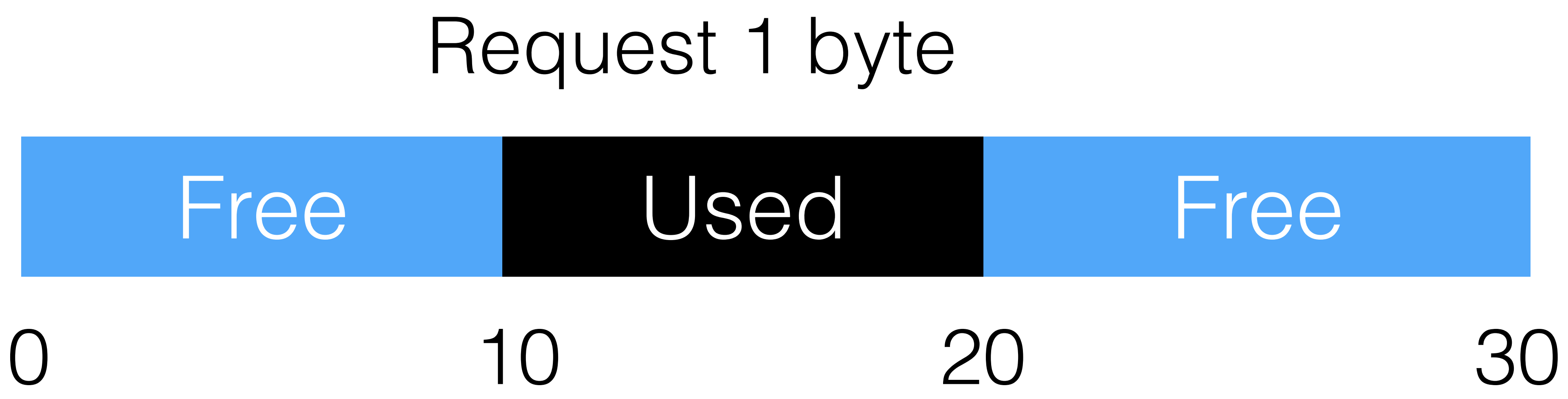
Split

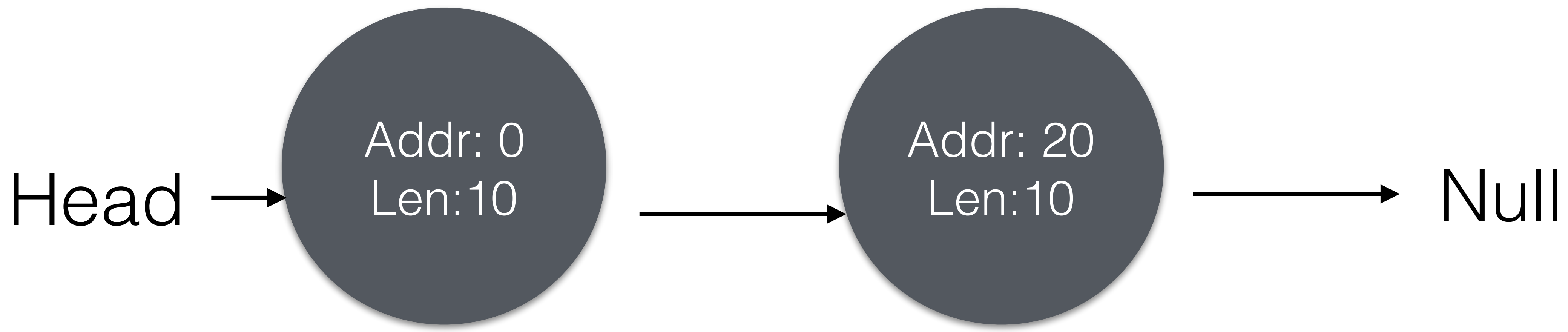

Request 1 byte
Free
Used
Free
0
10
20
30

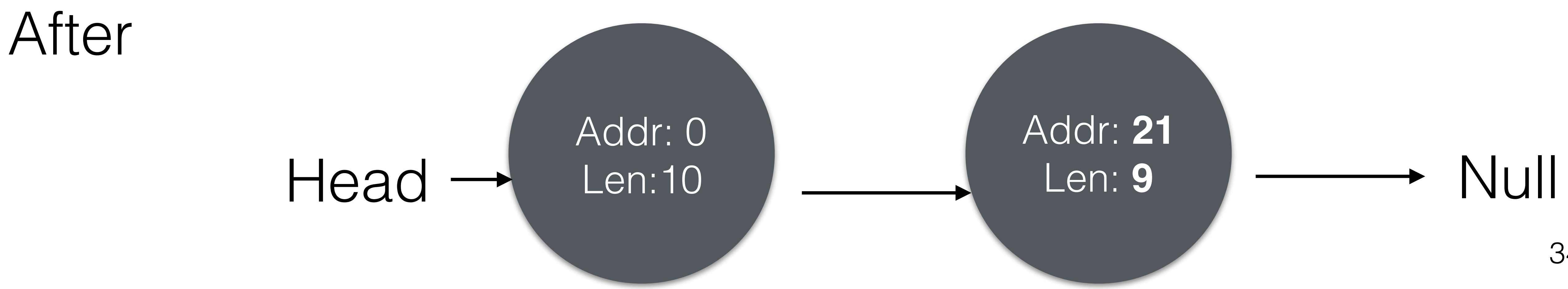

Head
Addr: 0
Len:10
Addr: 20
Len:10
Null

After

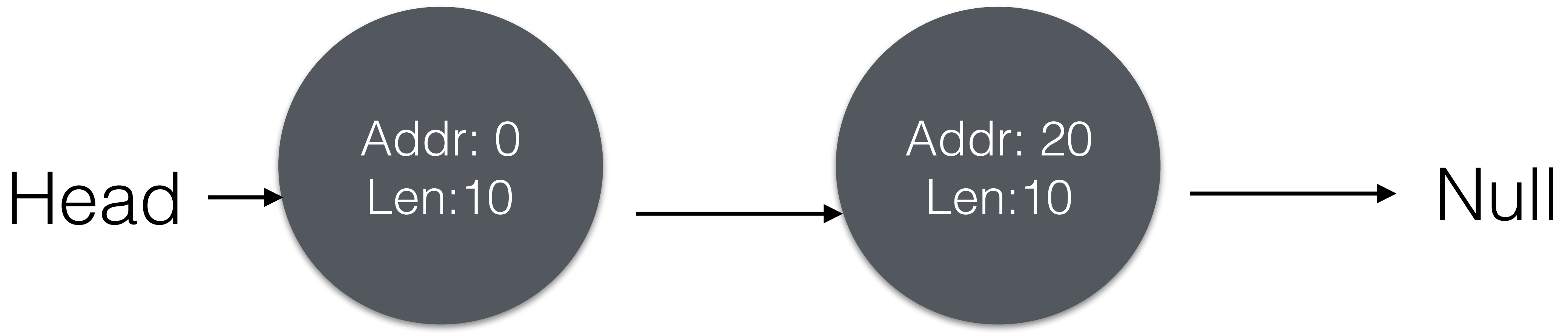
Split


Request 1 byte
Free
Used
Free
0
10
20
30

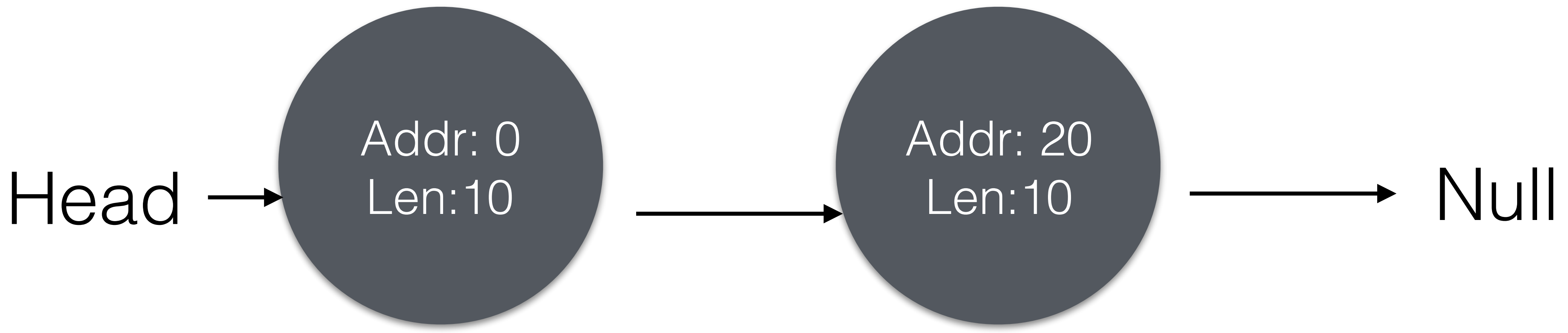

Head
Addr: 0
Len:10
Addr: 20
Len:10
Null


After
Head
Addr: 0
Len:10
Addr: **21**
Len: **9**
Null

Coalescing

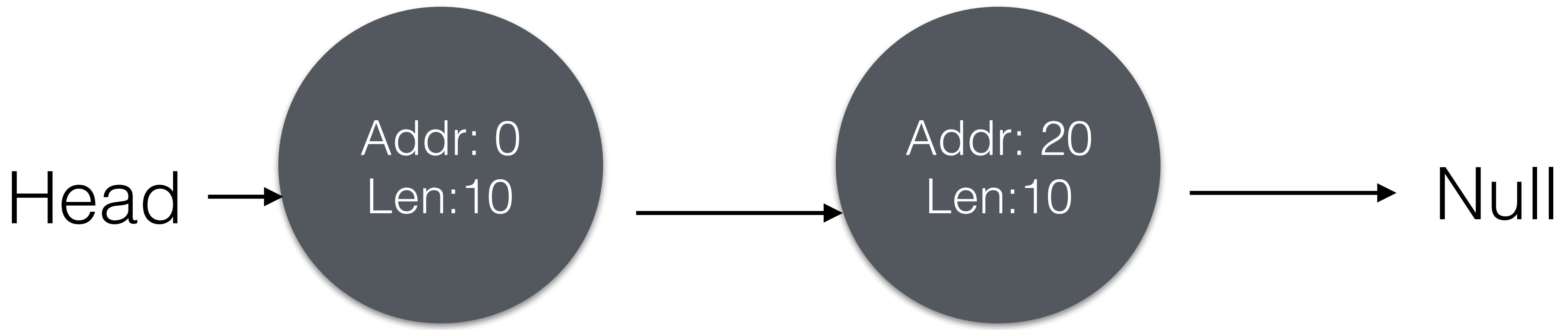


Coalescing

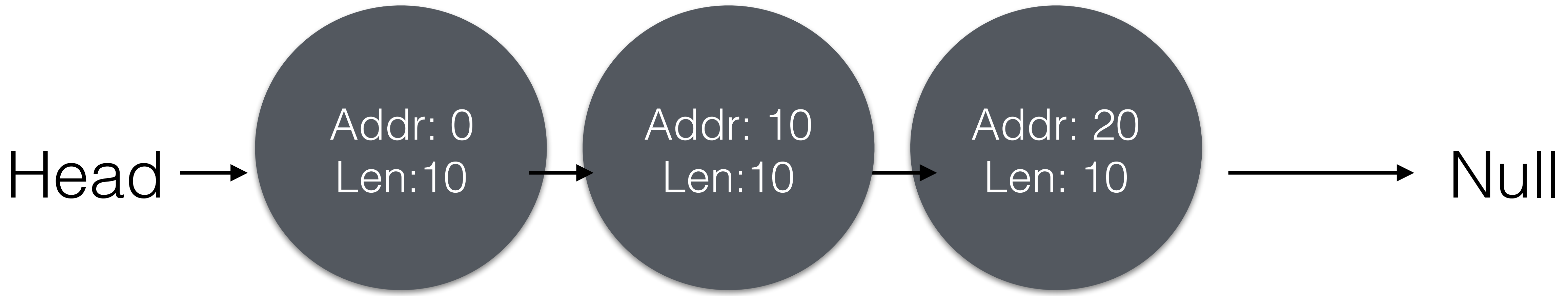


Free 10 bytes

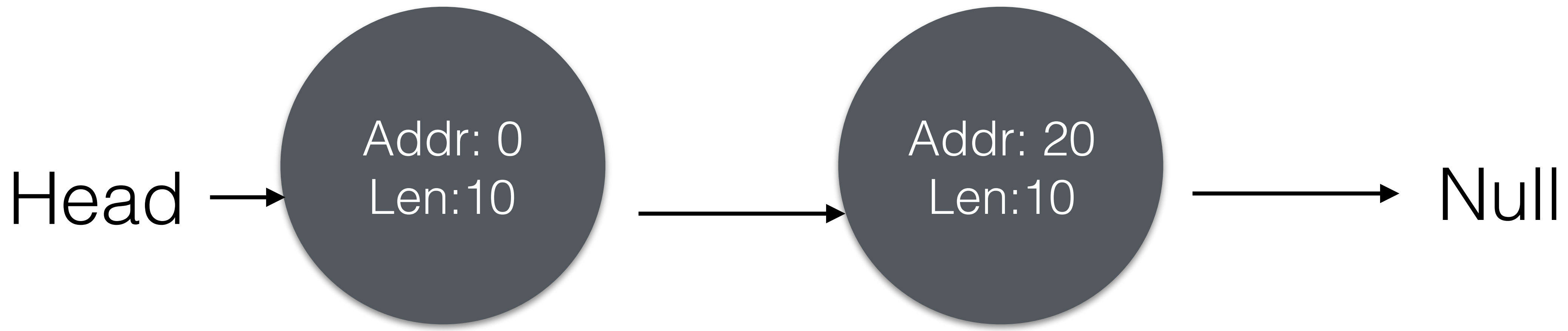
Coalescing



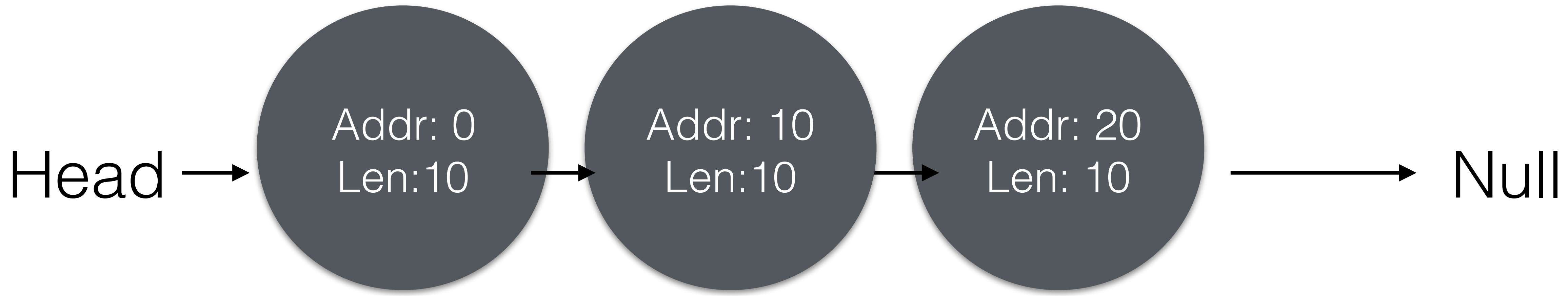
Free 10 bytes



Coalescing

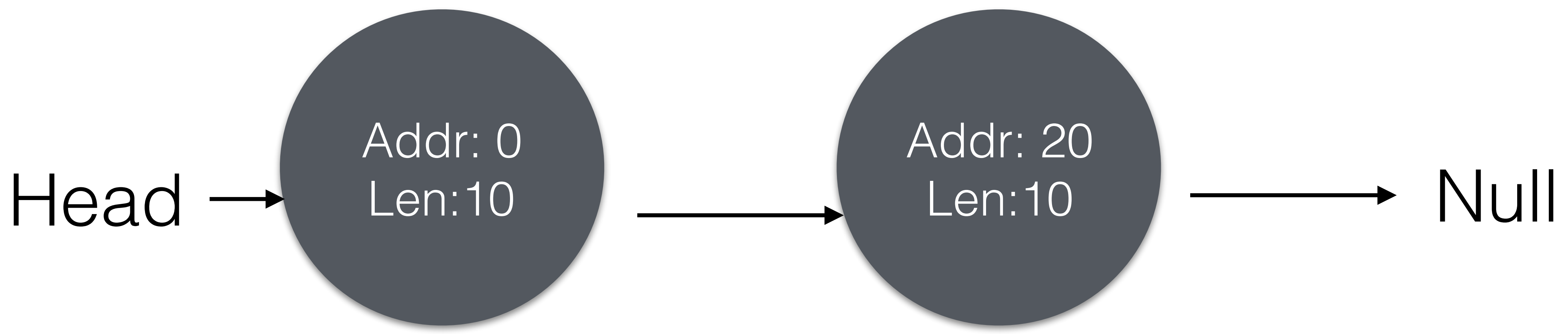


Free 10 bytes

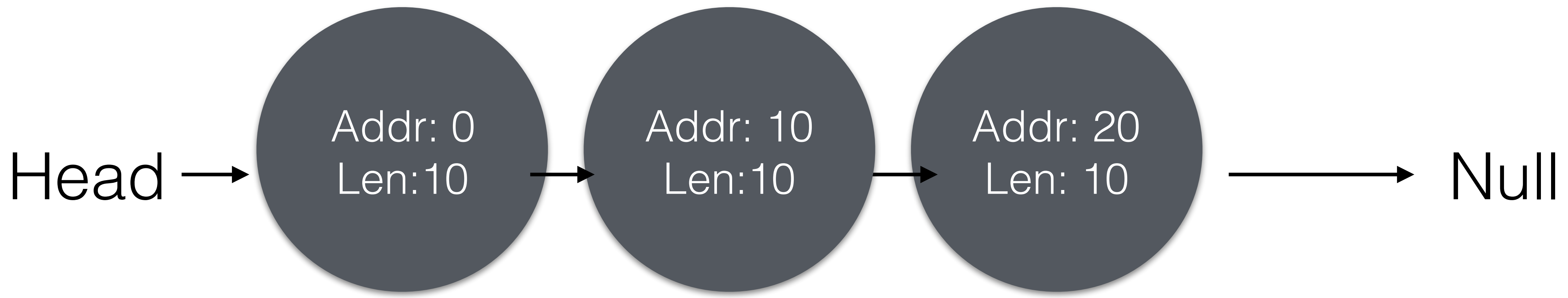


Coalesce

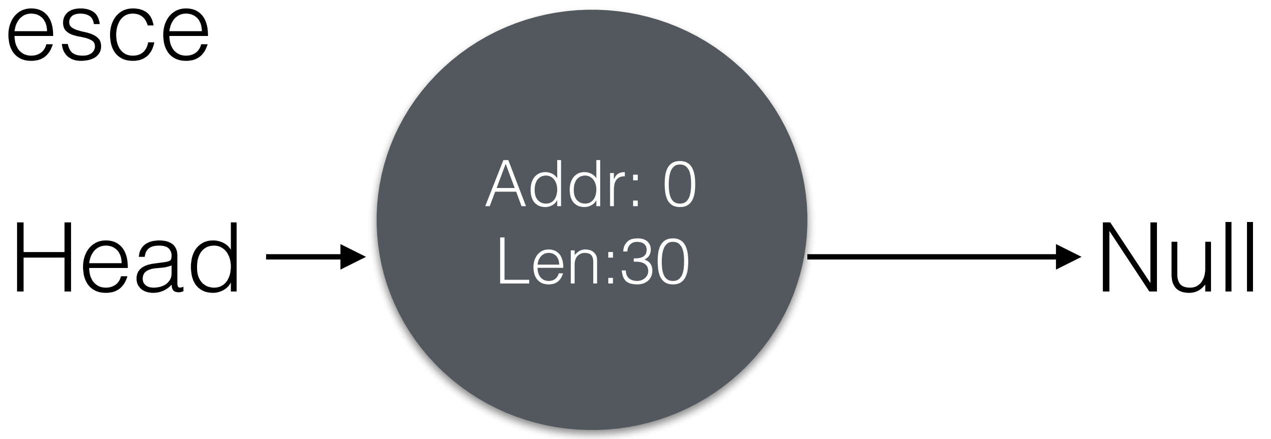
Coalescing



Free 10 bytes



Coalesce

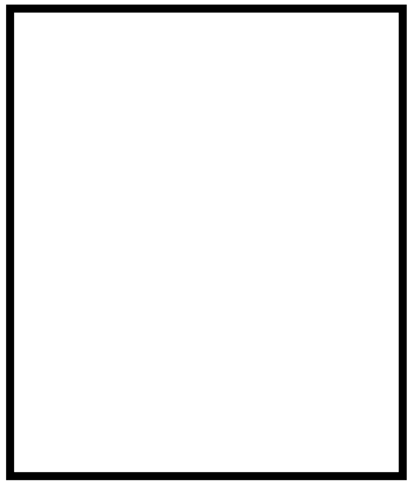


Tracking size of allocations

- Freeing —> give space back to heap

Tracking size of allocations

- Freeing —> give space back to heap



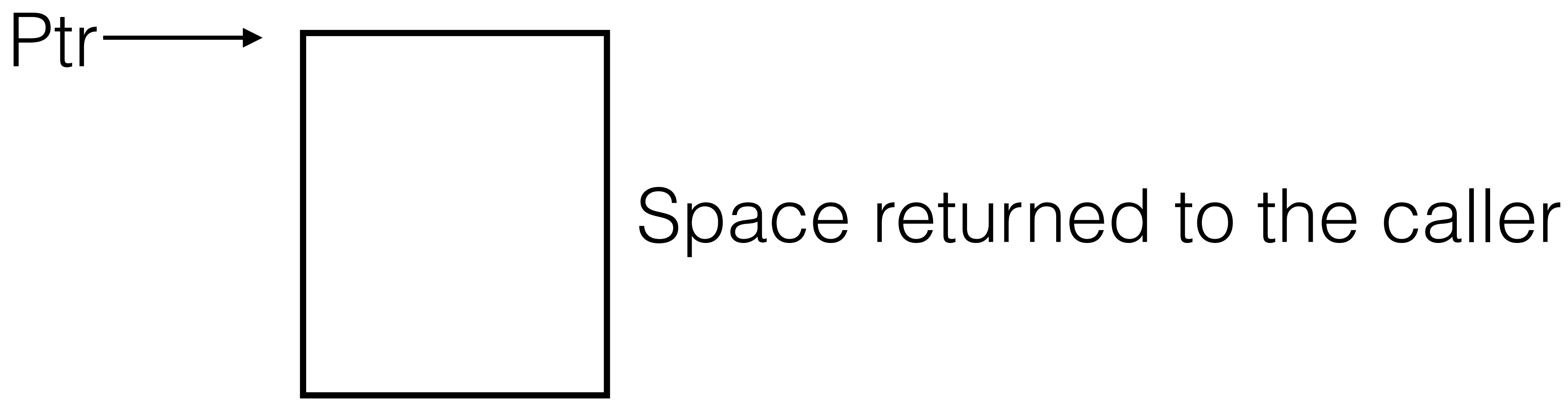
Tracking size of allocations

- Freeing —> give space back to heap

Space returned to the caller

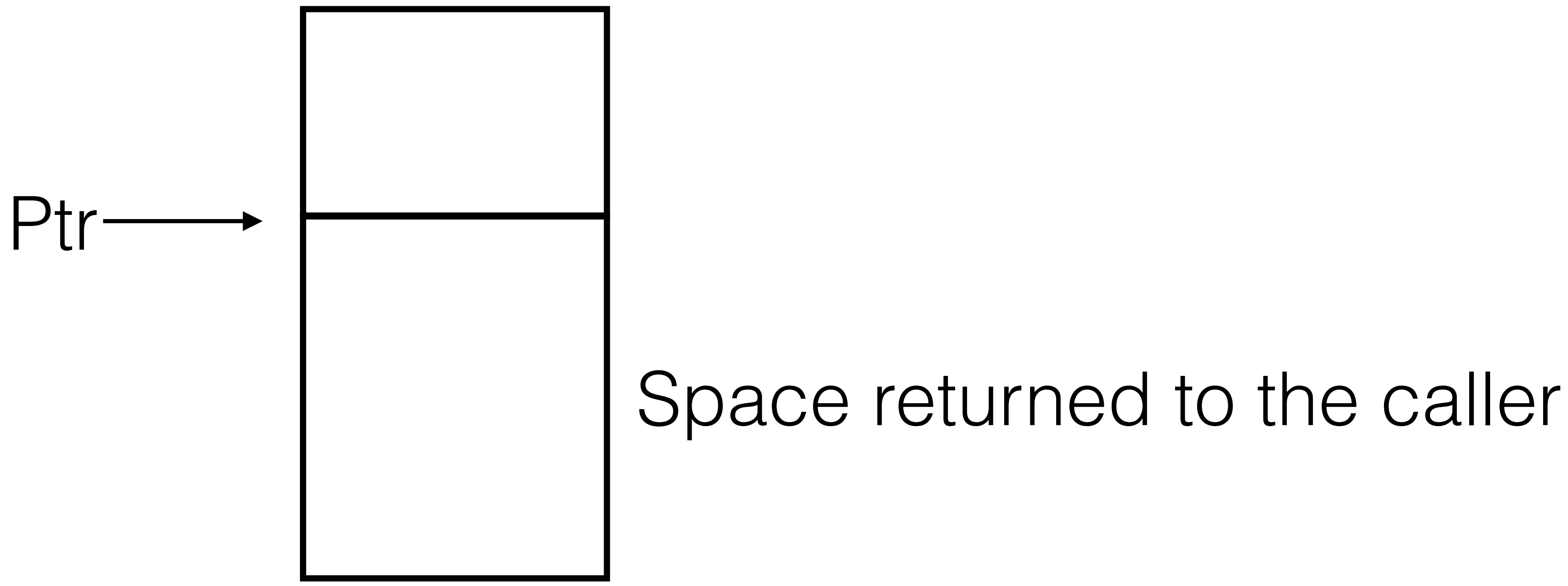
Tracking size of allocations

- Freeing —> give space back to heap



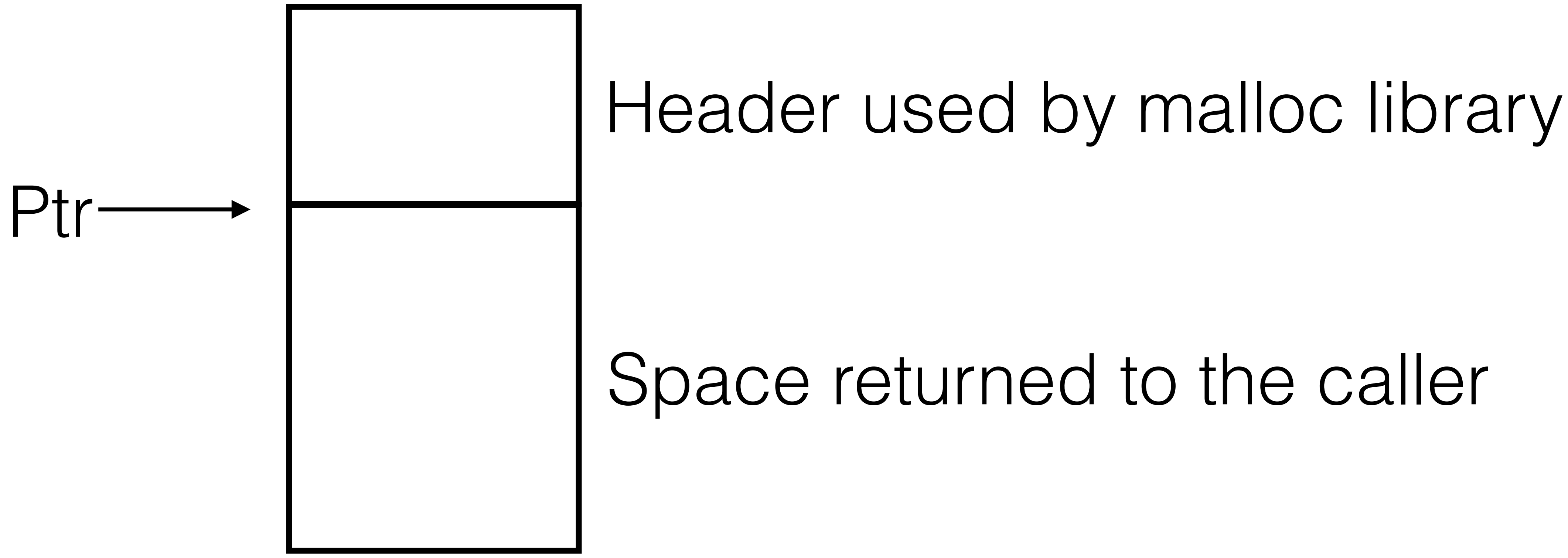
Tracking size of allocations

- Freeing —> give space back to heap



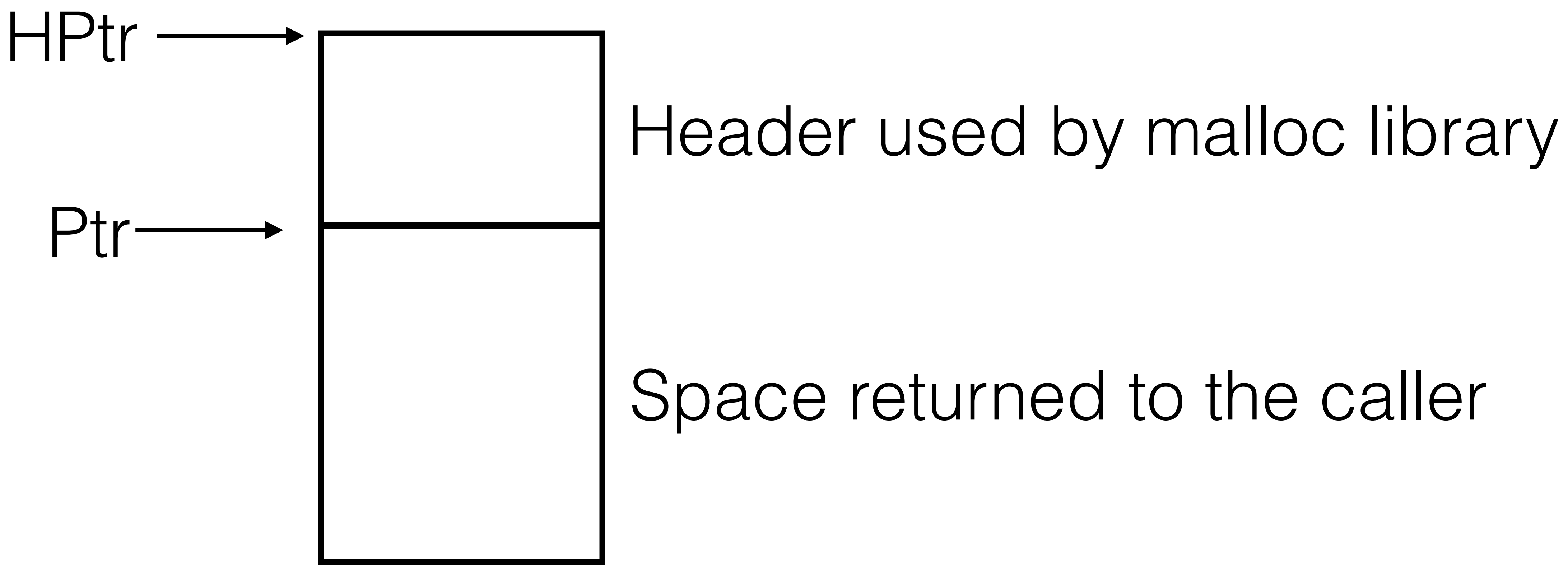
Tracking size of allocations

- Freeing —> give space back to heap



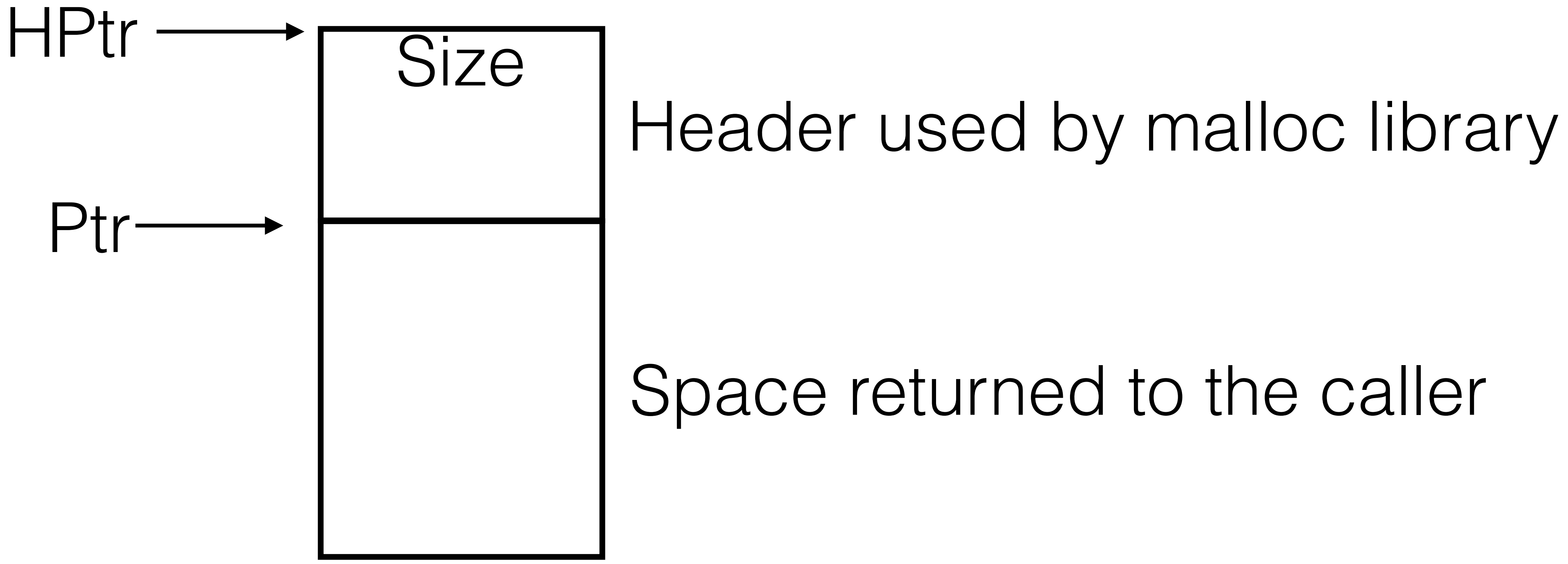
Tracking size of allocations

- Freeing —> give space back to heap



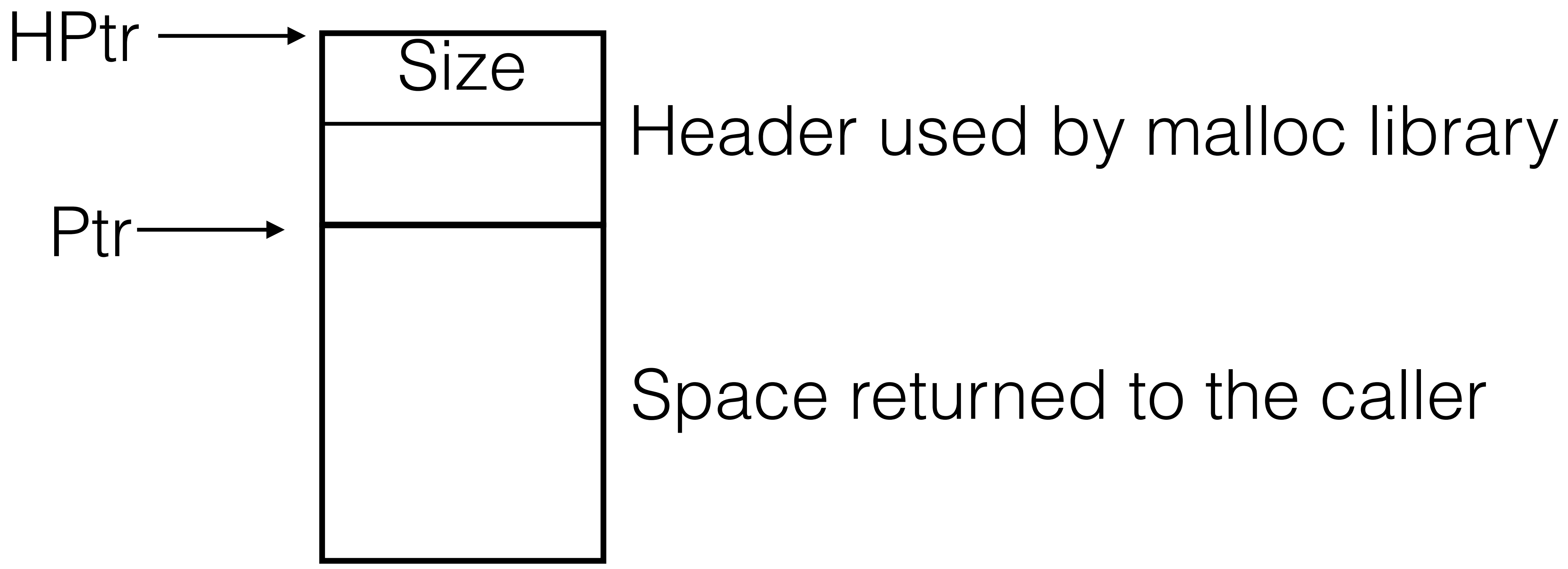
Tracking size of allocations

- Freeing —> give space back to heap



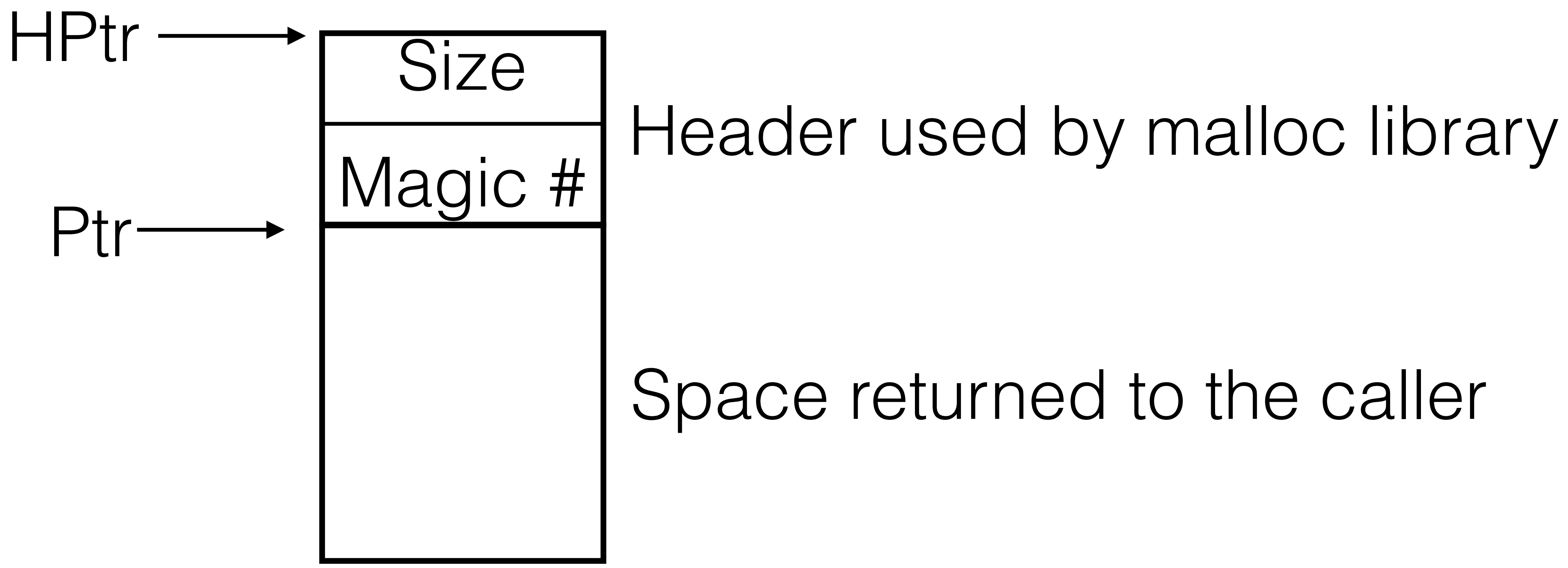
Tracking size of allocations

- Freeing —> give space back to heap

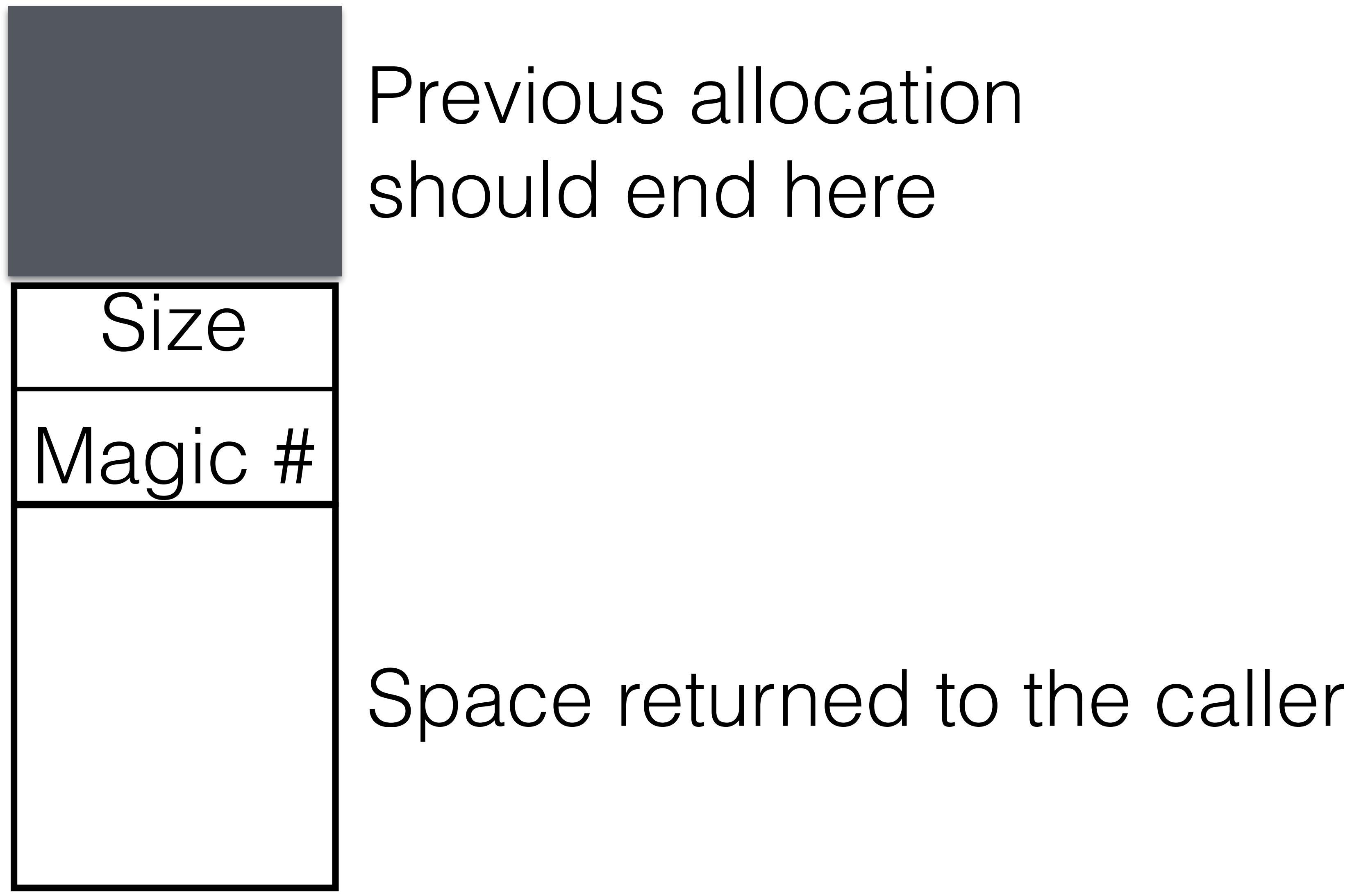


Tracking size of allocations

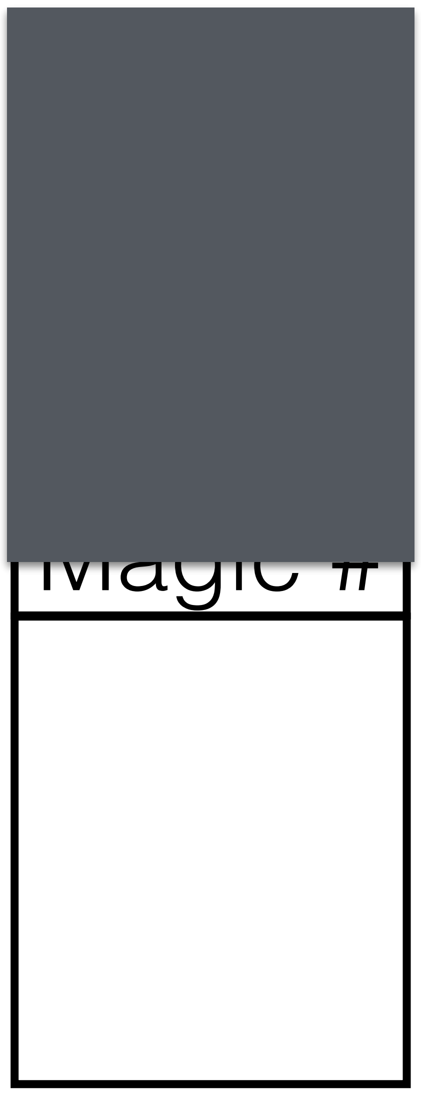
- Freeing —> give space back to heap



Why magic numbers?


Previous allocation should end here
Size
Magic #
Space returned to the caller

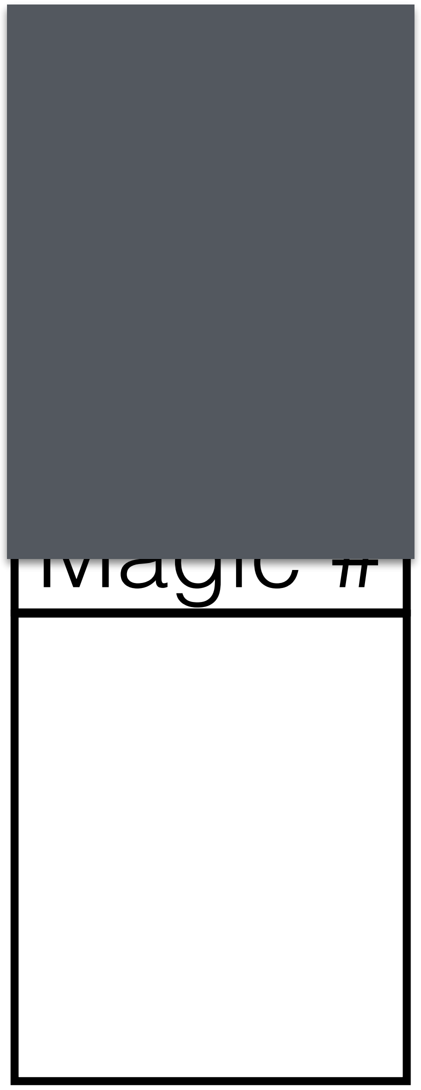
Why magic numbers?



But, instead ends here …

Space returned to the caller

Why magic numbers?



```
assert(hptr->magic == 2939239)
```

Space returned to the caller

Where else we use Magic numbers?

Let’s use hexdump

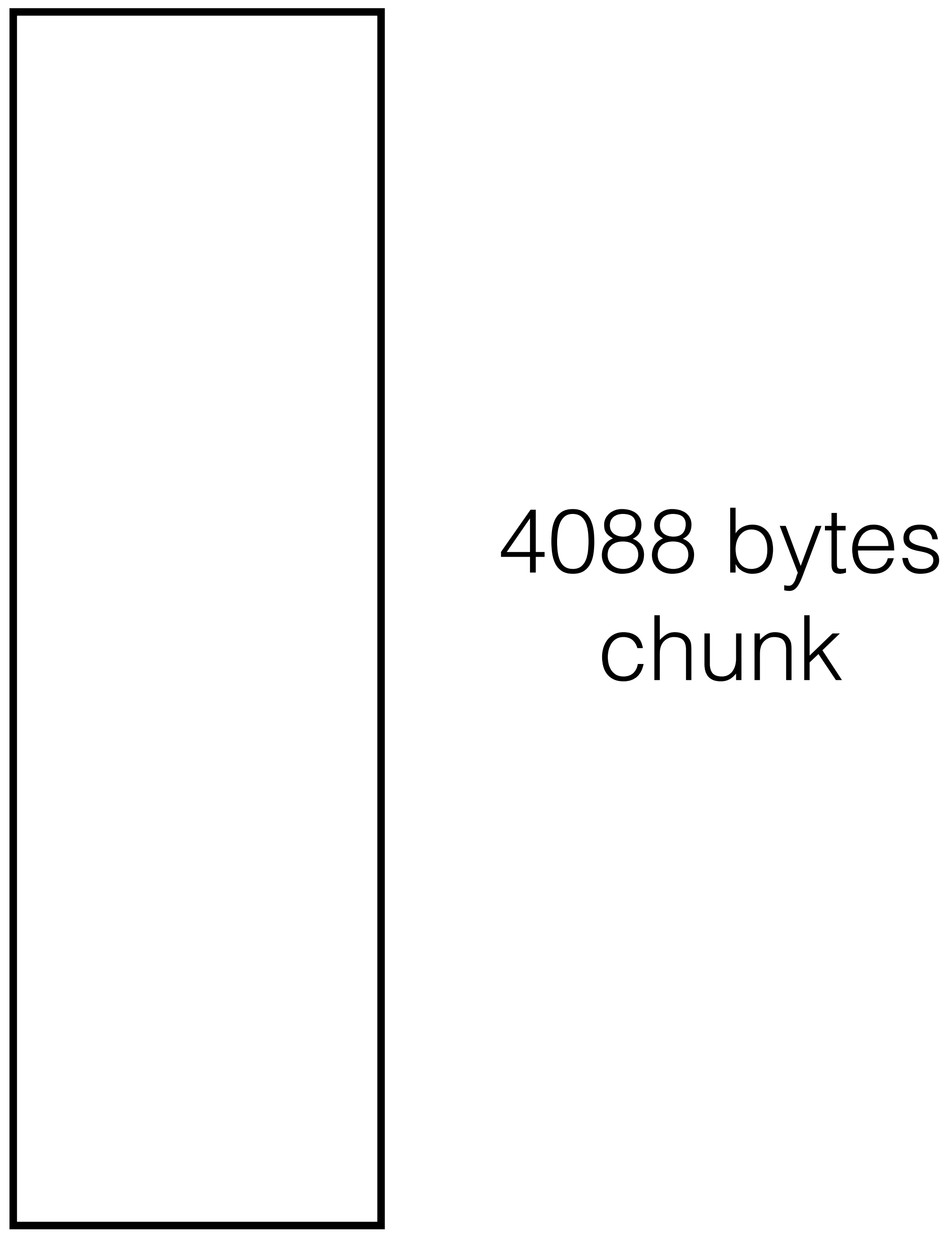
Example

Unallocated
4KB heap

4088 bytes
chunk

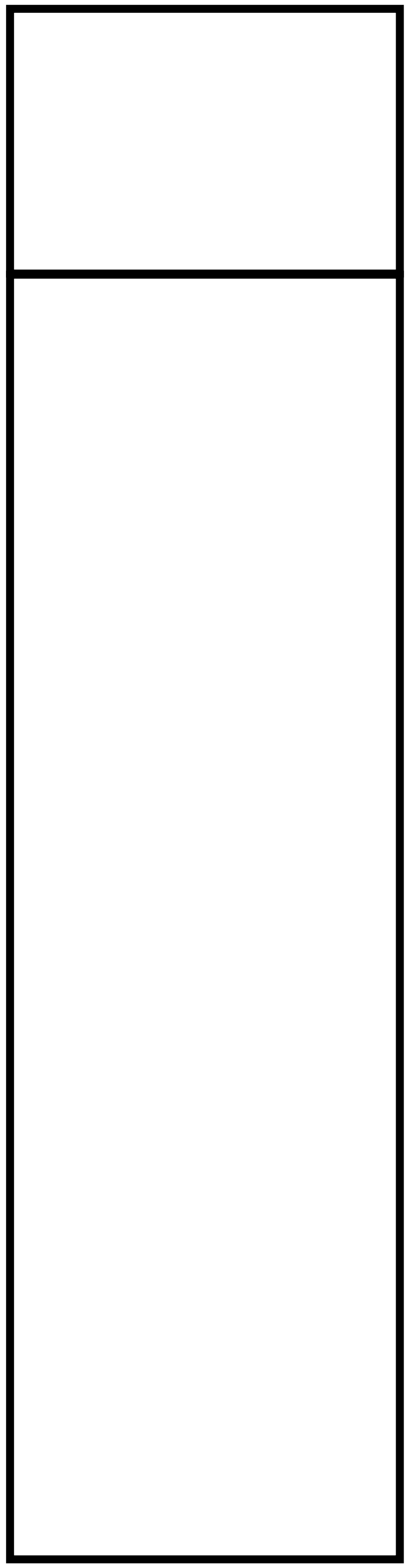
Example

Unallocated
4KB heap



Example

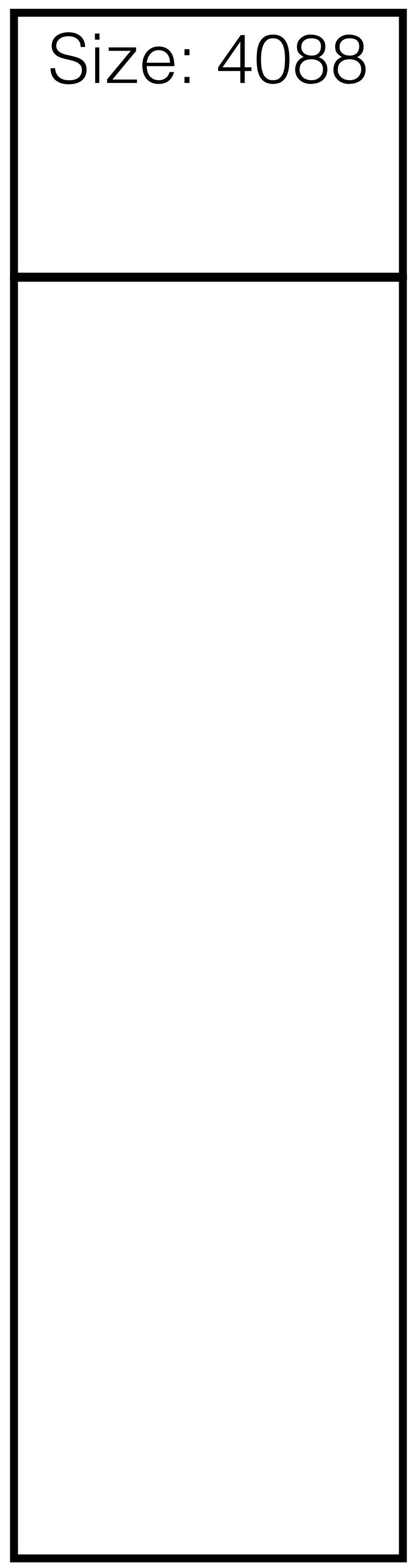
Unallocated
4KB heap



4088 bytes
chunk

Example

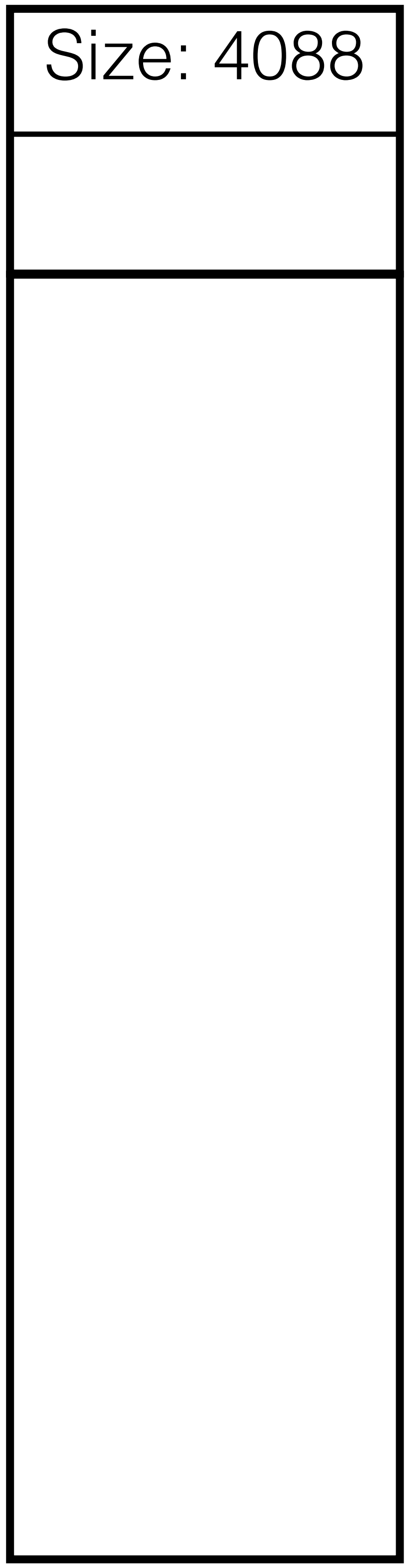
Unallocated
4KB heap



4088 bytes
chunk

Example

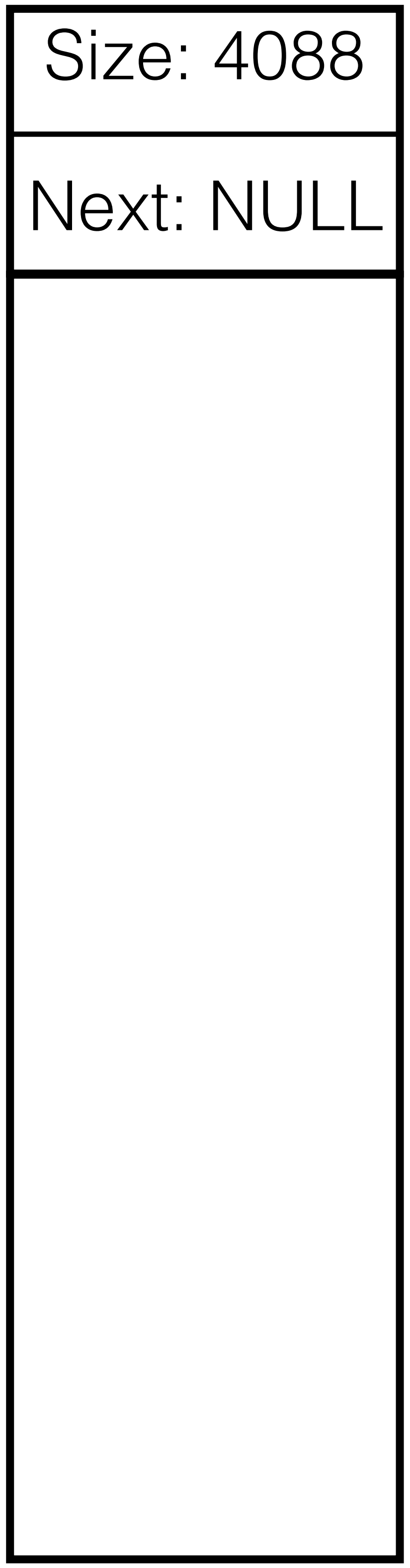
Unallocated 4KB heap



4088 bytes chunk

Example

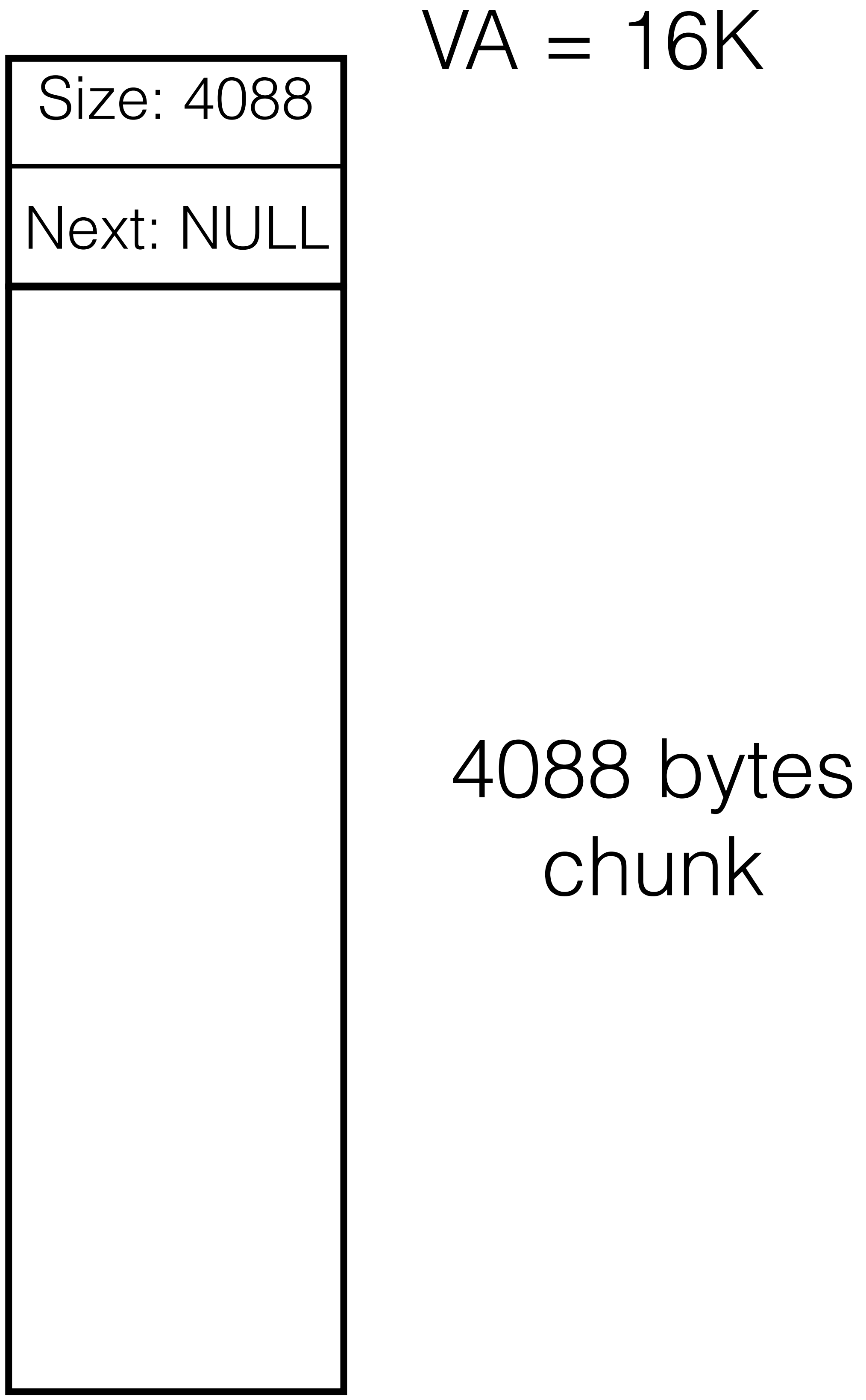
Unallocated
4KB heap



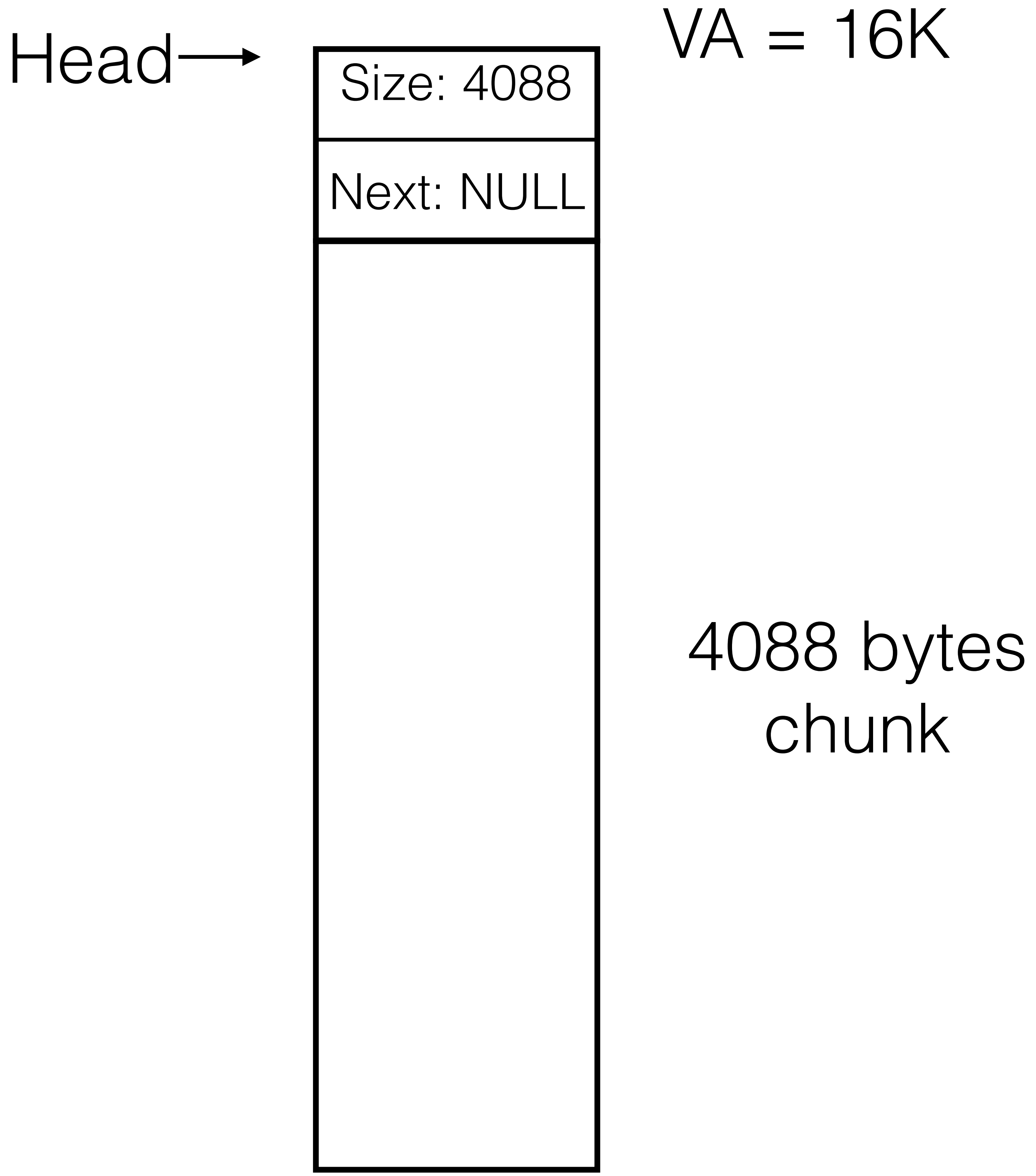
4088 bytes
chunk

Example

Unallocated 4KB heap



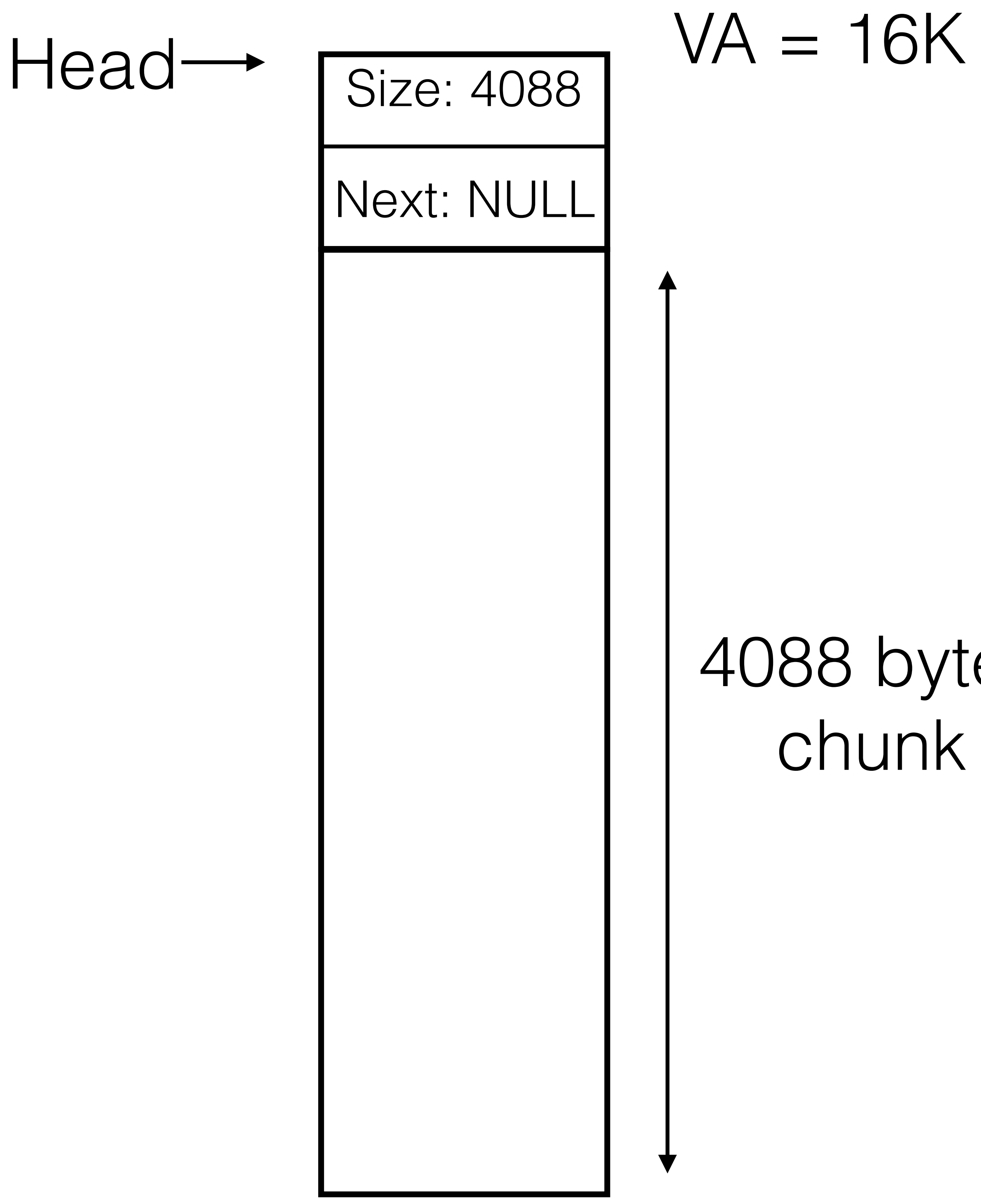
Example



Unallocated 4KB heap

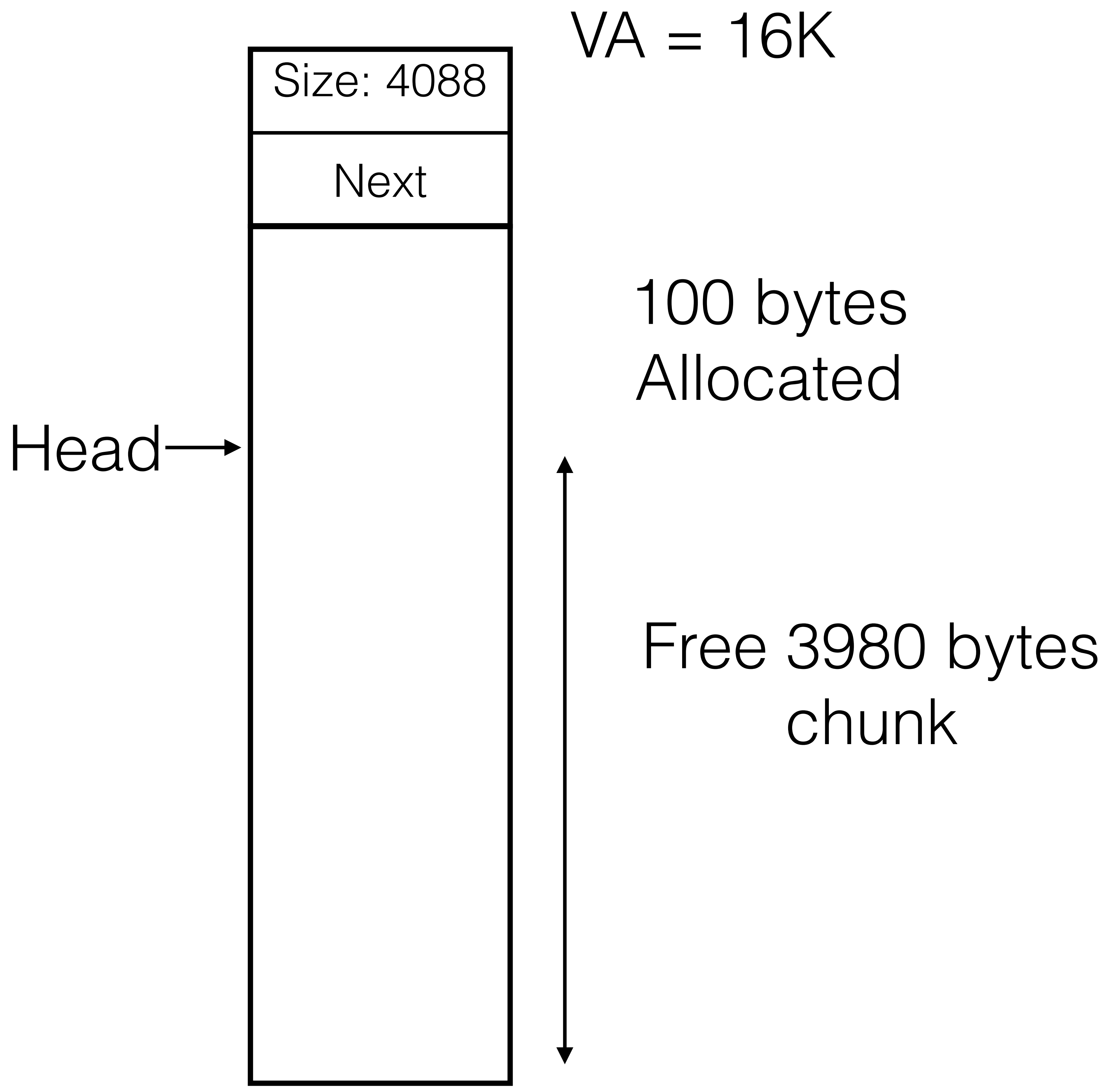
Example

Unallocated 4KB heap



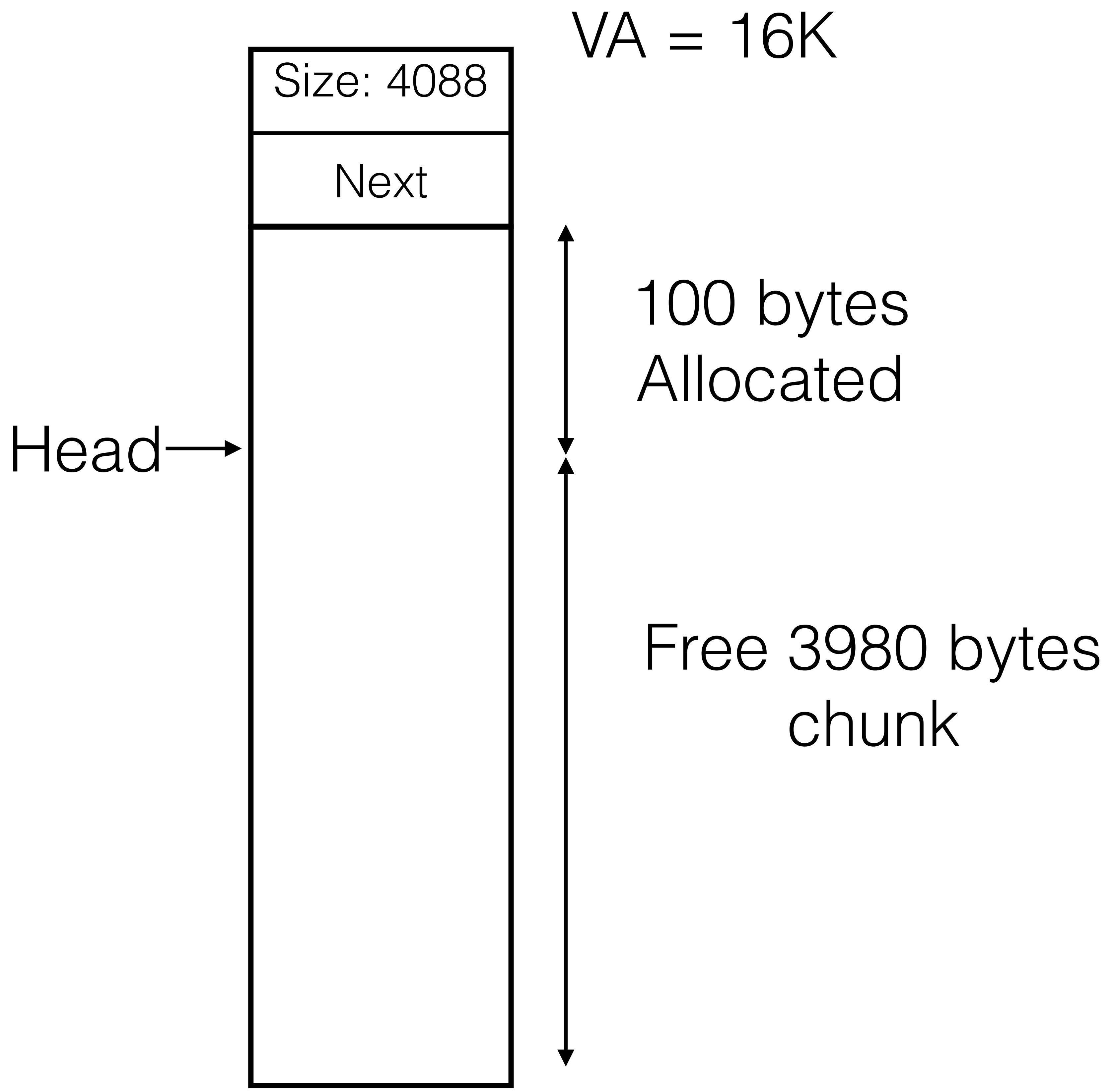
Example

After 1 allocation

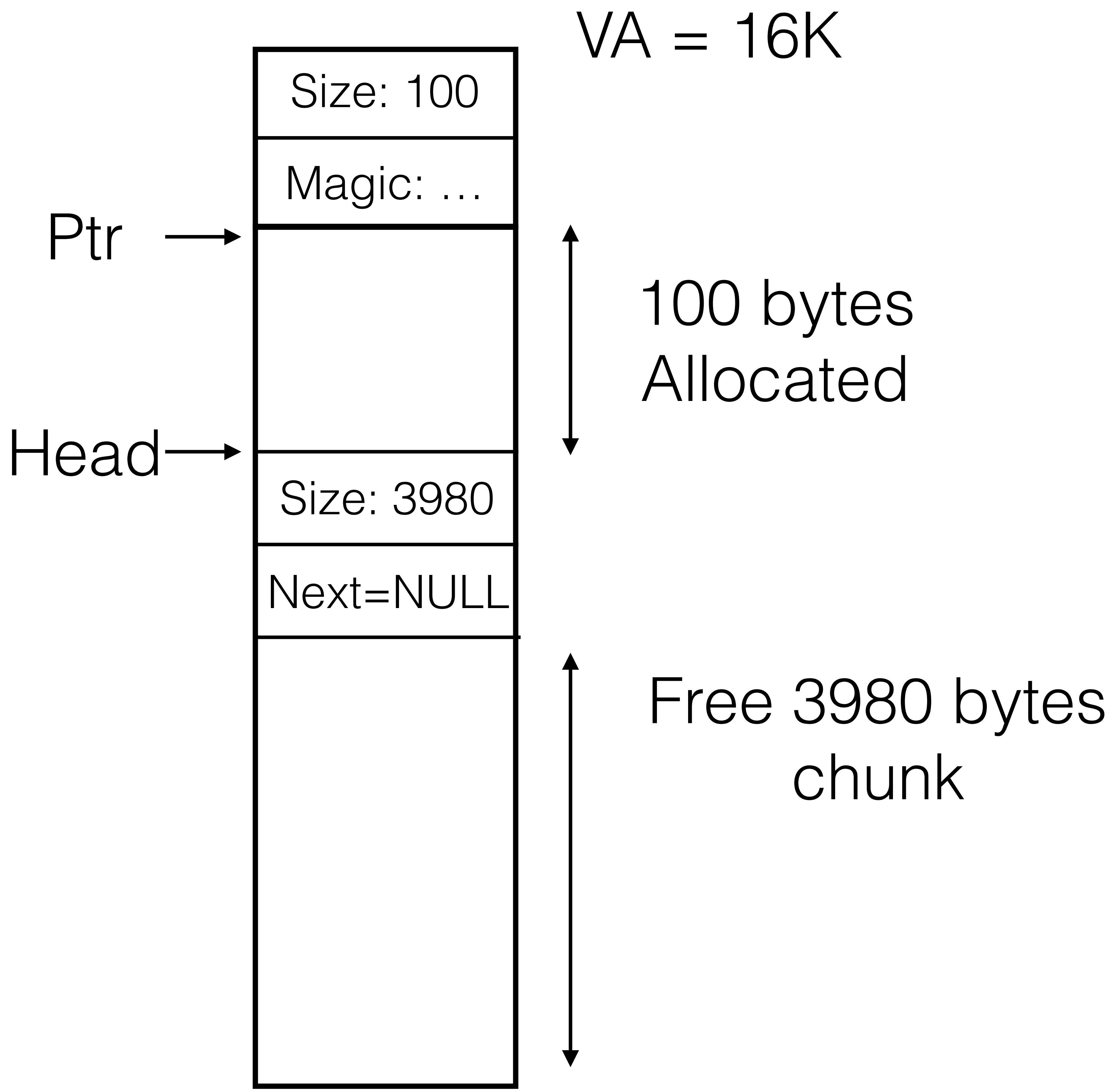


Example

After 1 allocation



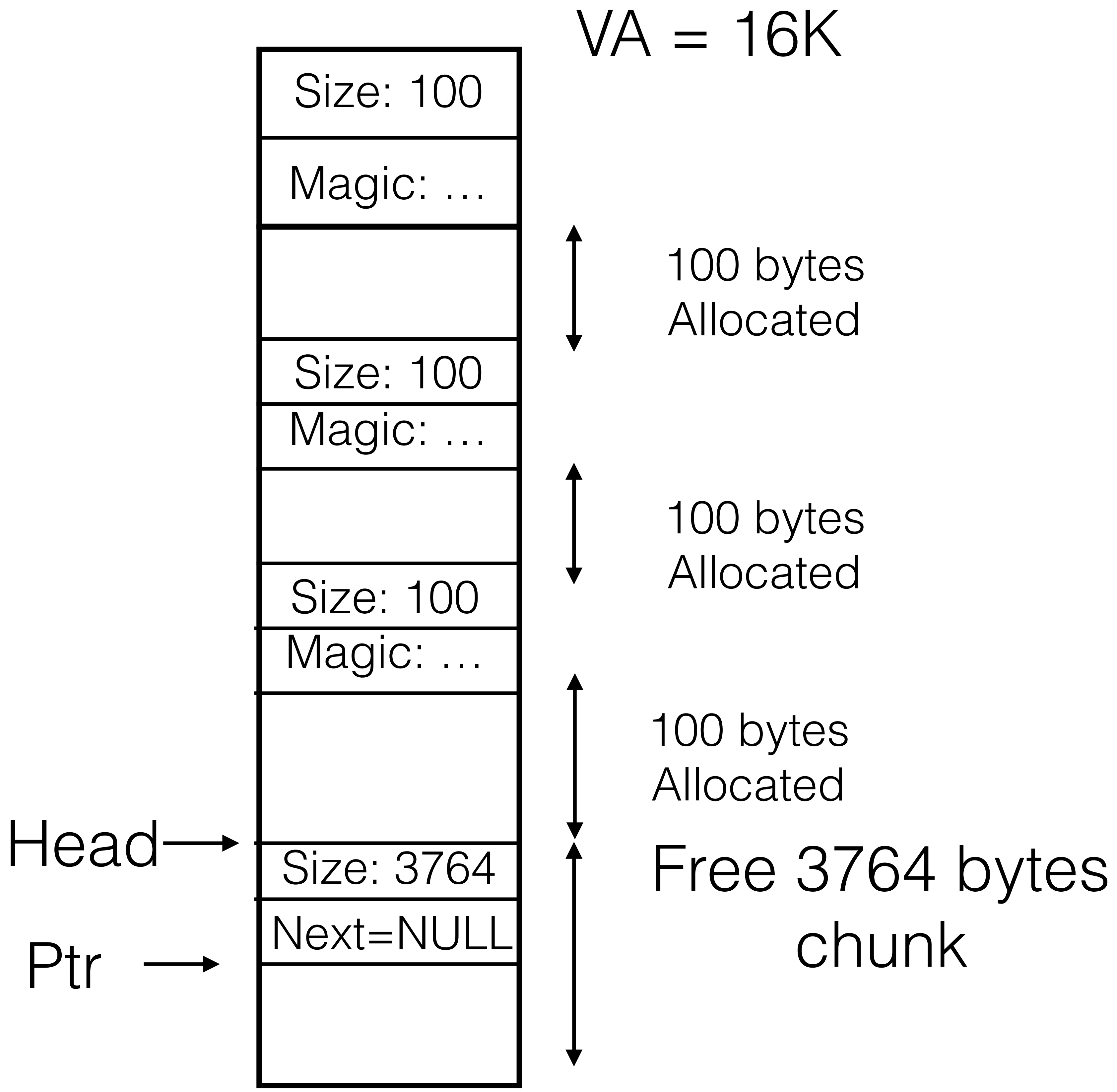
Example



After 1 allocation

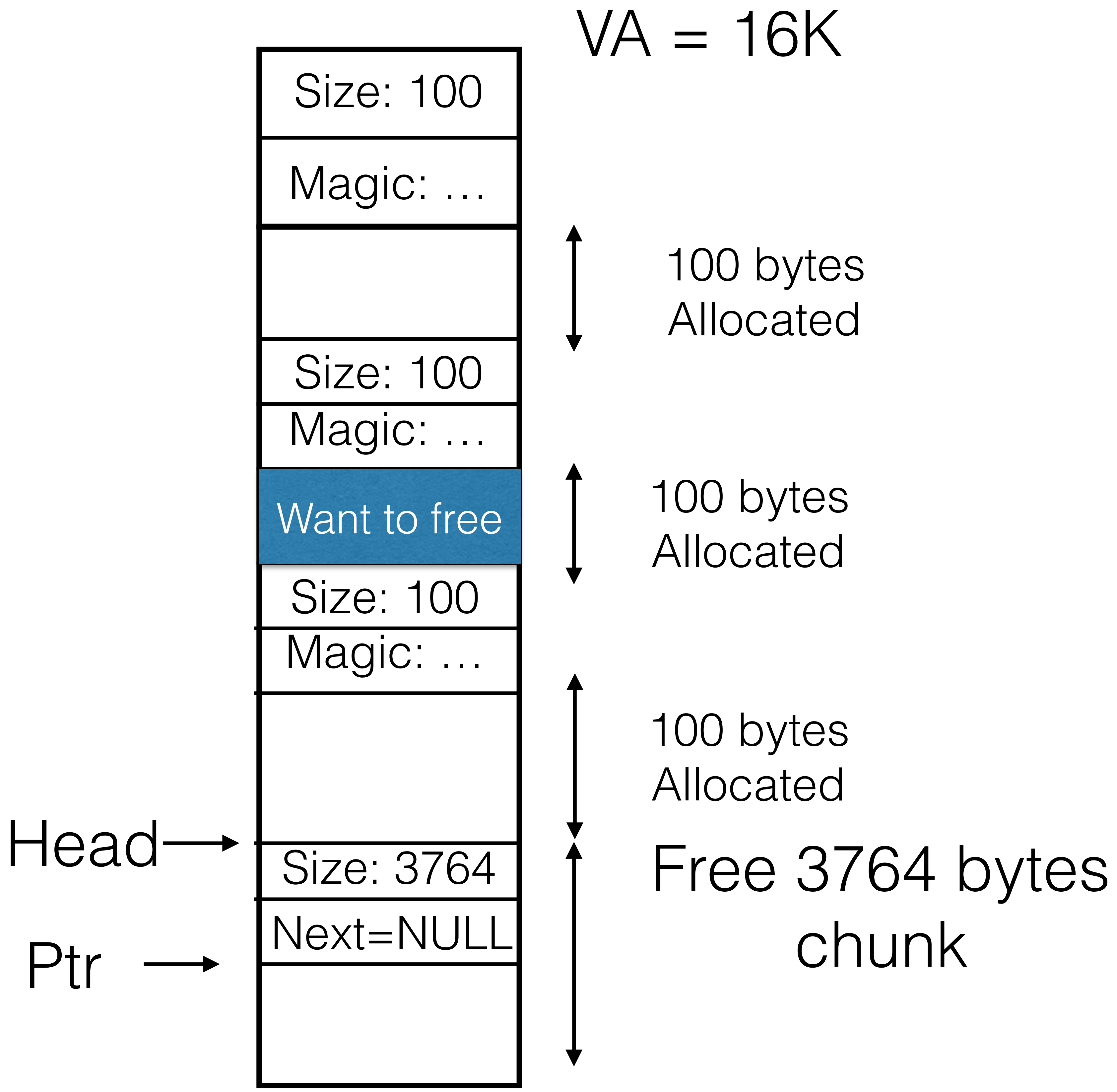
Example

After 3 Allocations



Example

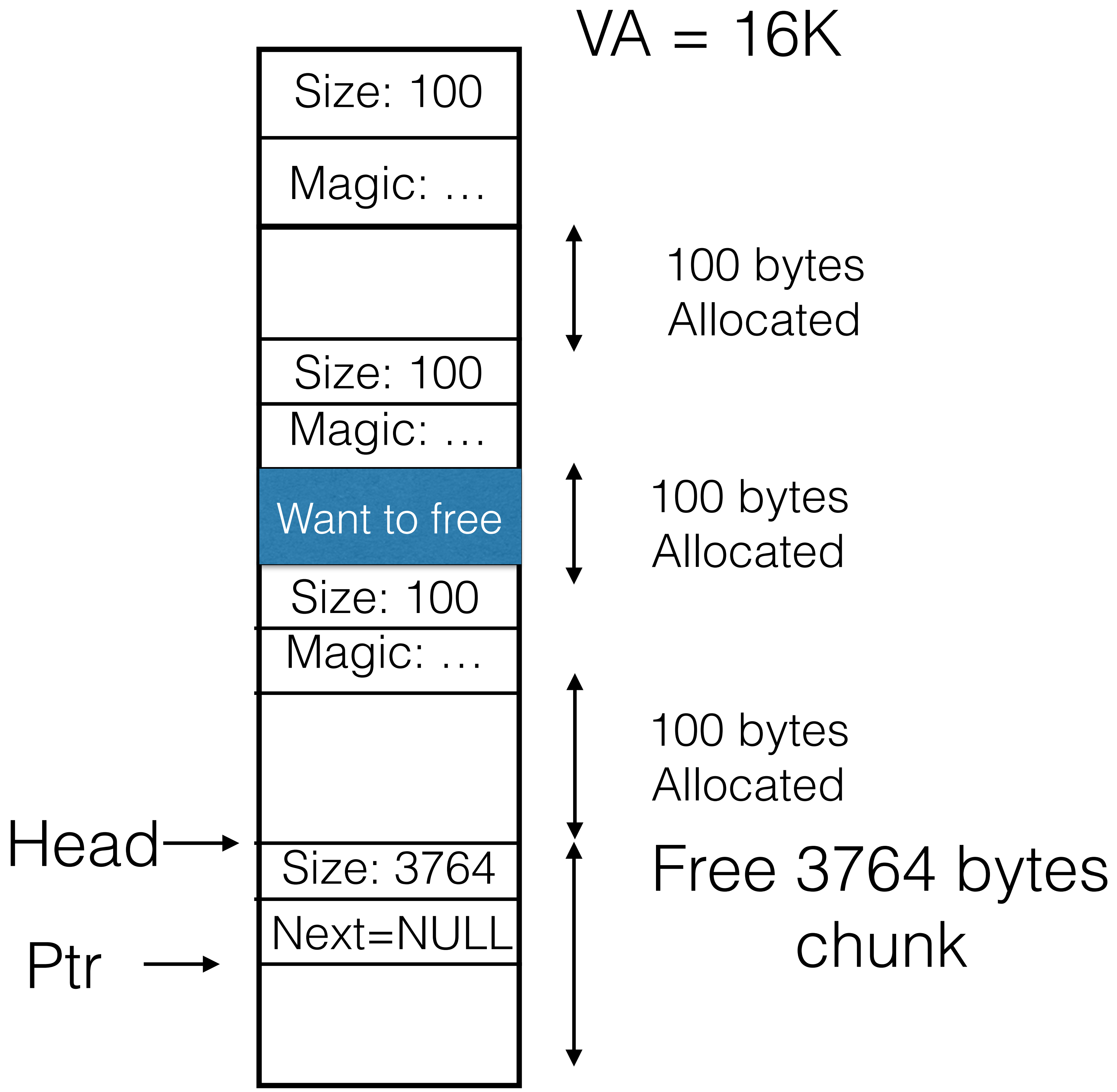
Freeing an allocation



Example

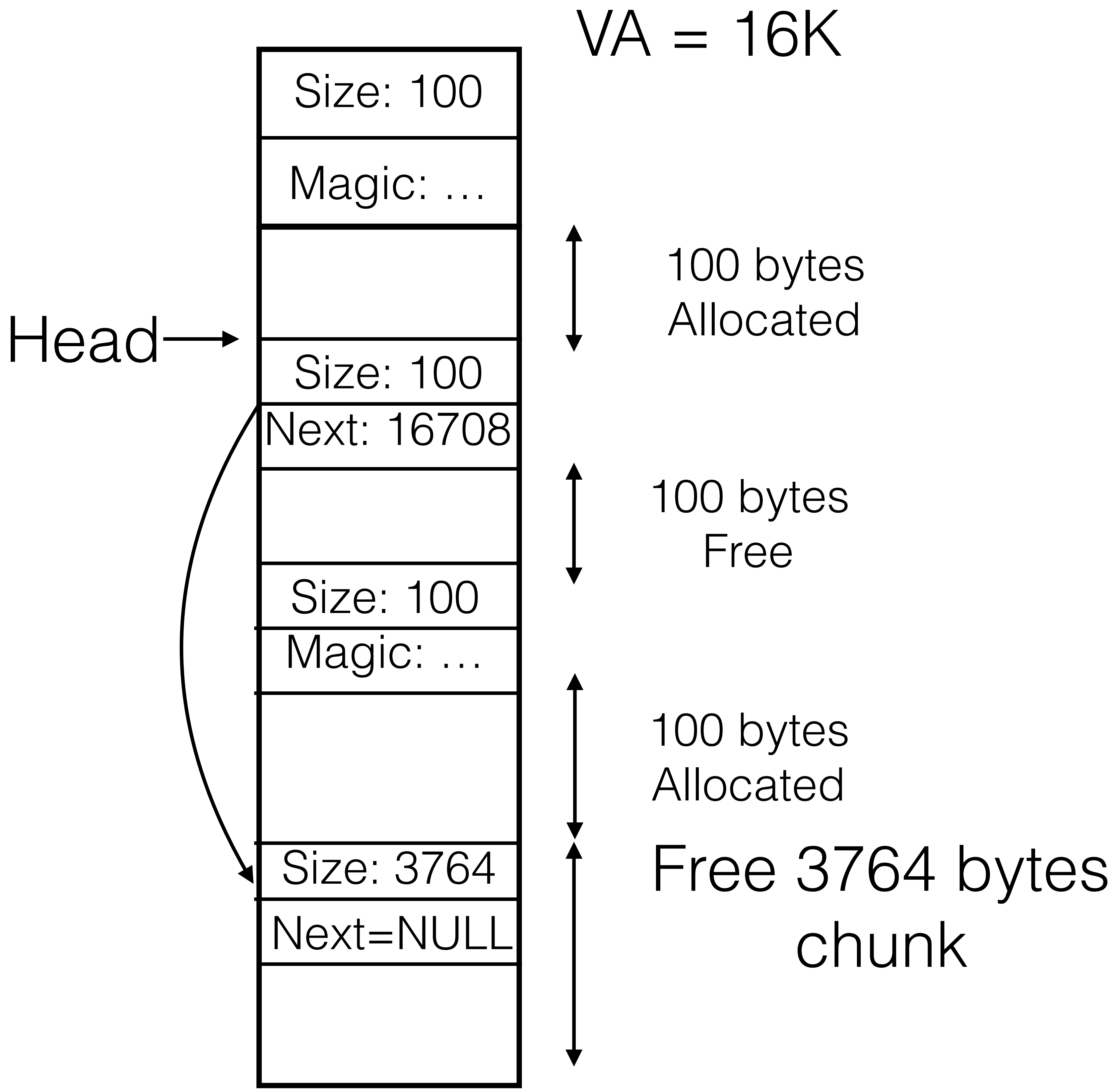
Freeing an allocation

Free(16K + 8 + 100 + 8)
= Free(16384 + 116)
= Free(16500)



Example

Freeing an allocation



Allocation Strategies

Allocation Strategies

Allocation Strategies

Before

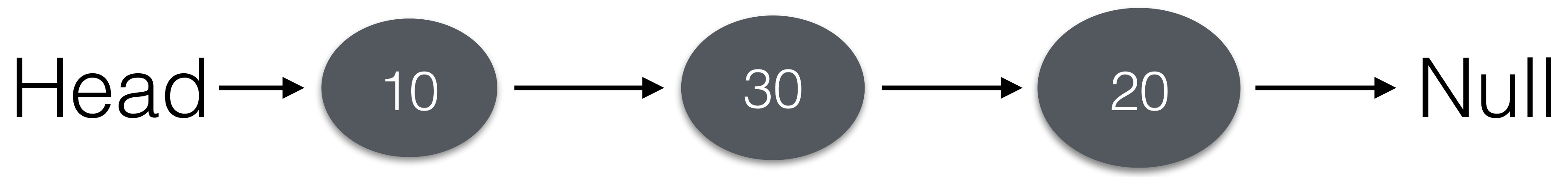
Allocation Strategies

Before

$$\begin{tikzcd}
\text{Head} \arrow[r] & 10 \arrow[r] & 30 \arrow[r] & 20 \arrow[r] & \text{Null}
\end{tikzcd}$$

Allocation Strategies

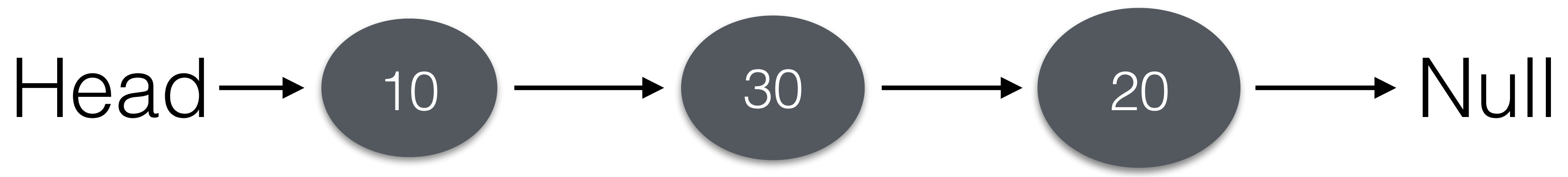
Before



Allocate 15 bytes

Allocation Strategies

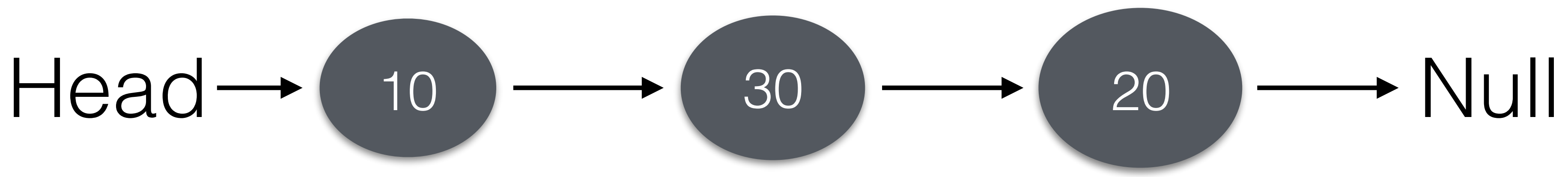
Before



Allocate 15 bytes

Allocation Strategies

Before



Allocate 15 bytes

Allocation Strategies

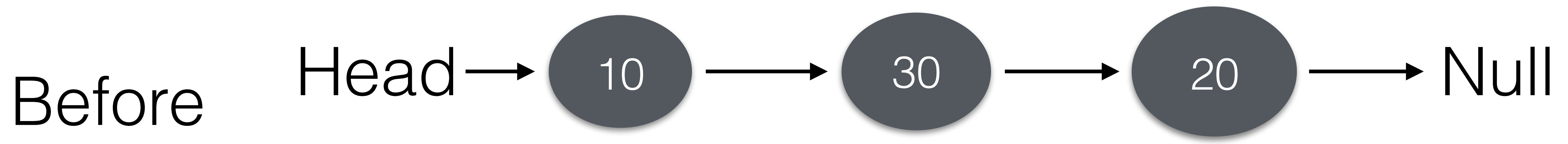
Before

$$\begin{tikzcd}\text{Head} \arrow[r] & 10 \arrow[r] & 30 \arrow[r] & 20 \arrow[r] & \text{Null}\end{tikzcd}$$

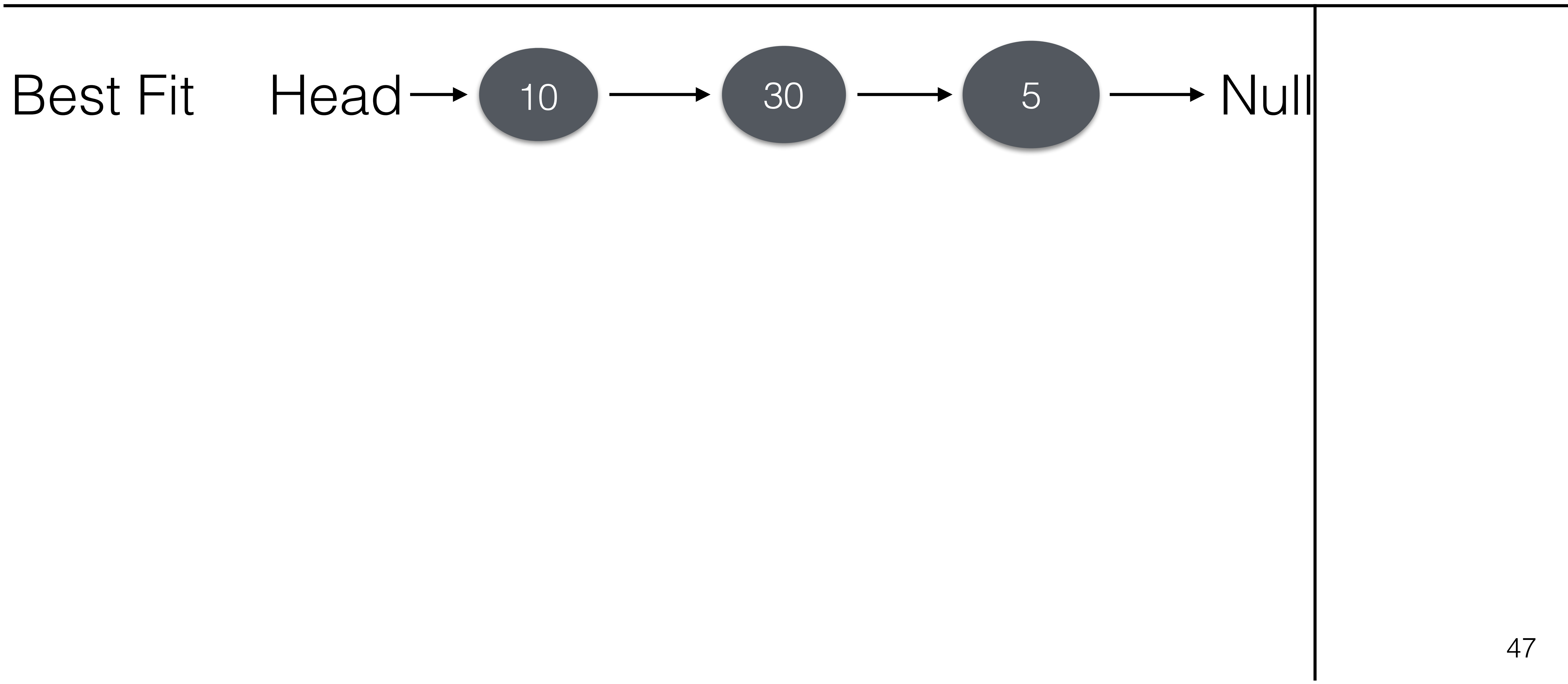
Allocate 15 bytes

Best Fit

Allocation Strategies

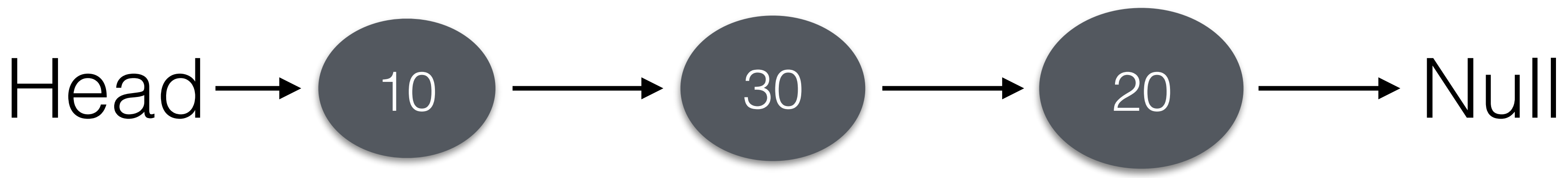


Allocate 15 bytes



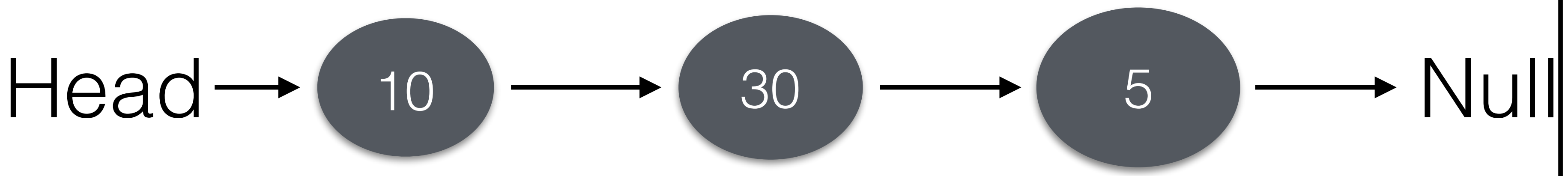
Allocation Strategies

Before



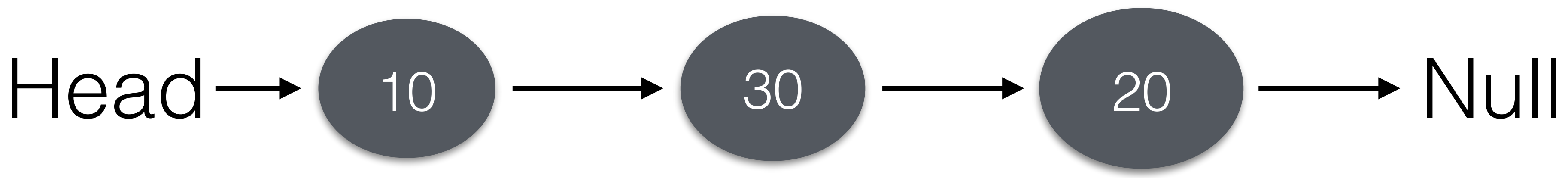
Allocate 15 bytes

Best Fit

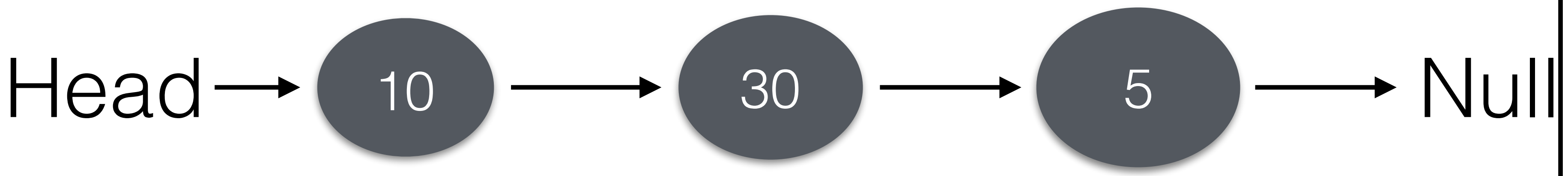


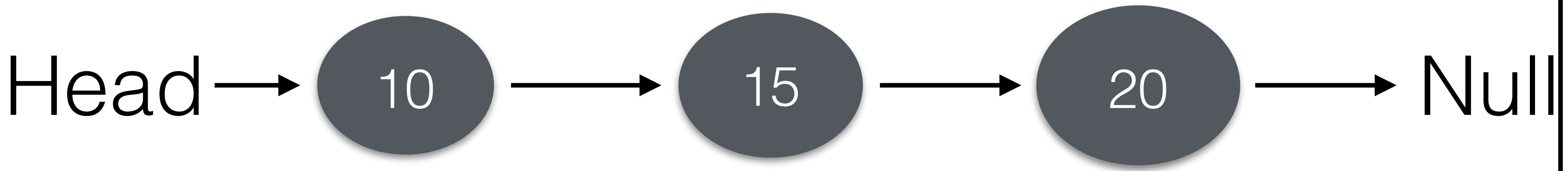
Worst Fit

Allocation Strategies

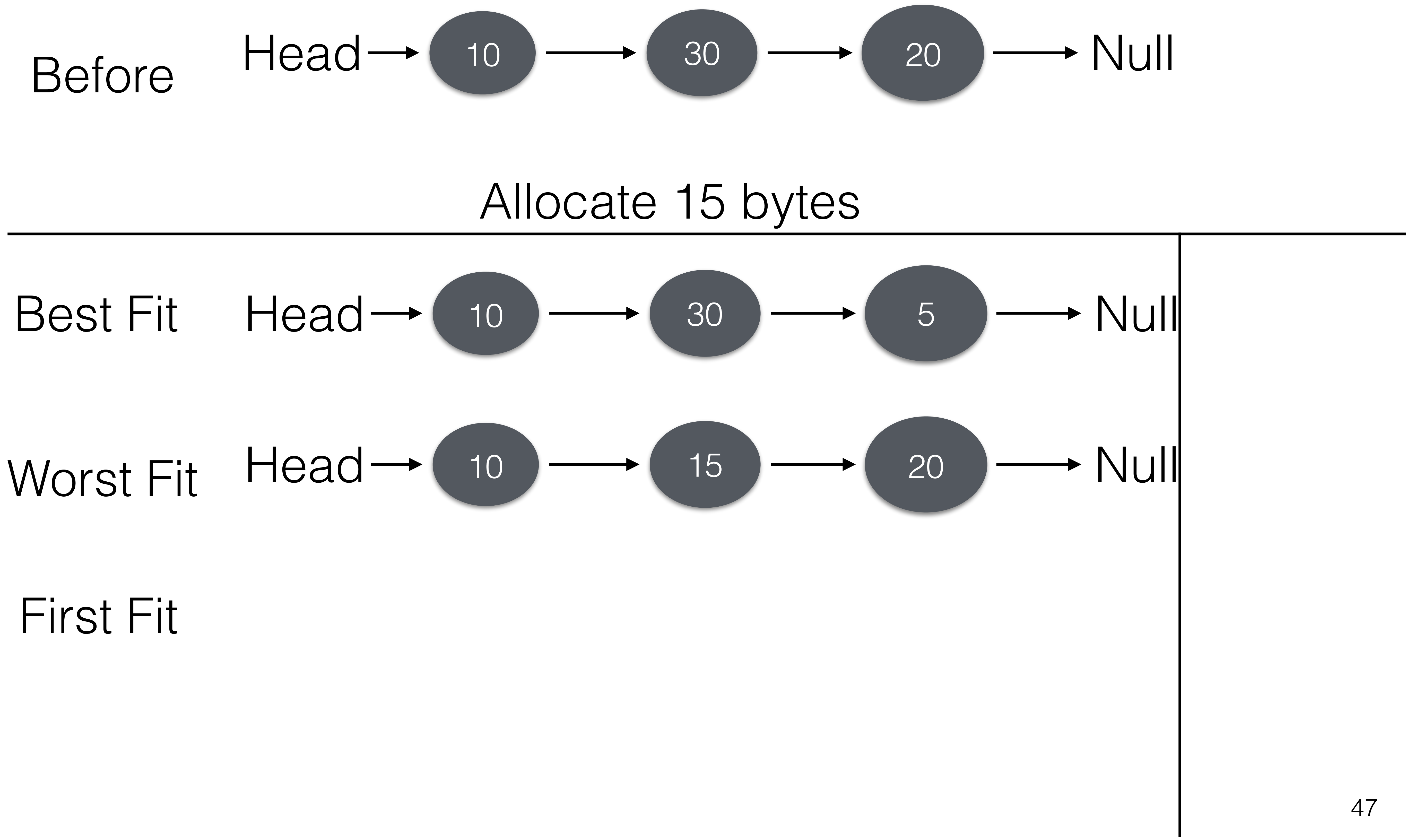
Before


Allocate 15 bytes

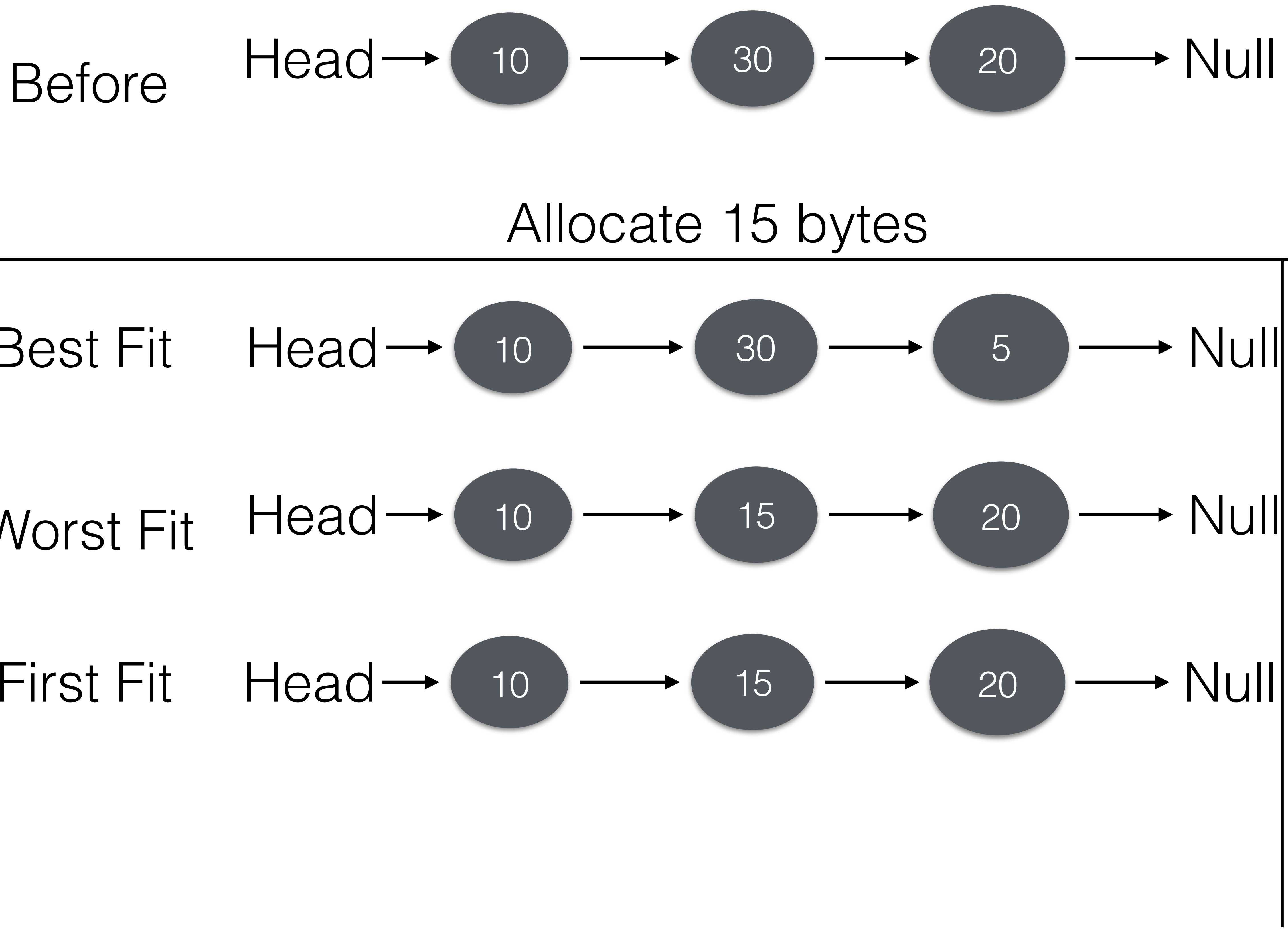
Best Fit


Worst Fit


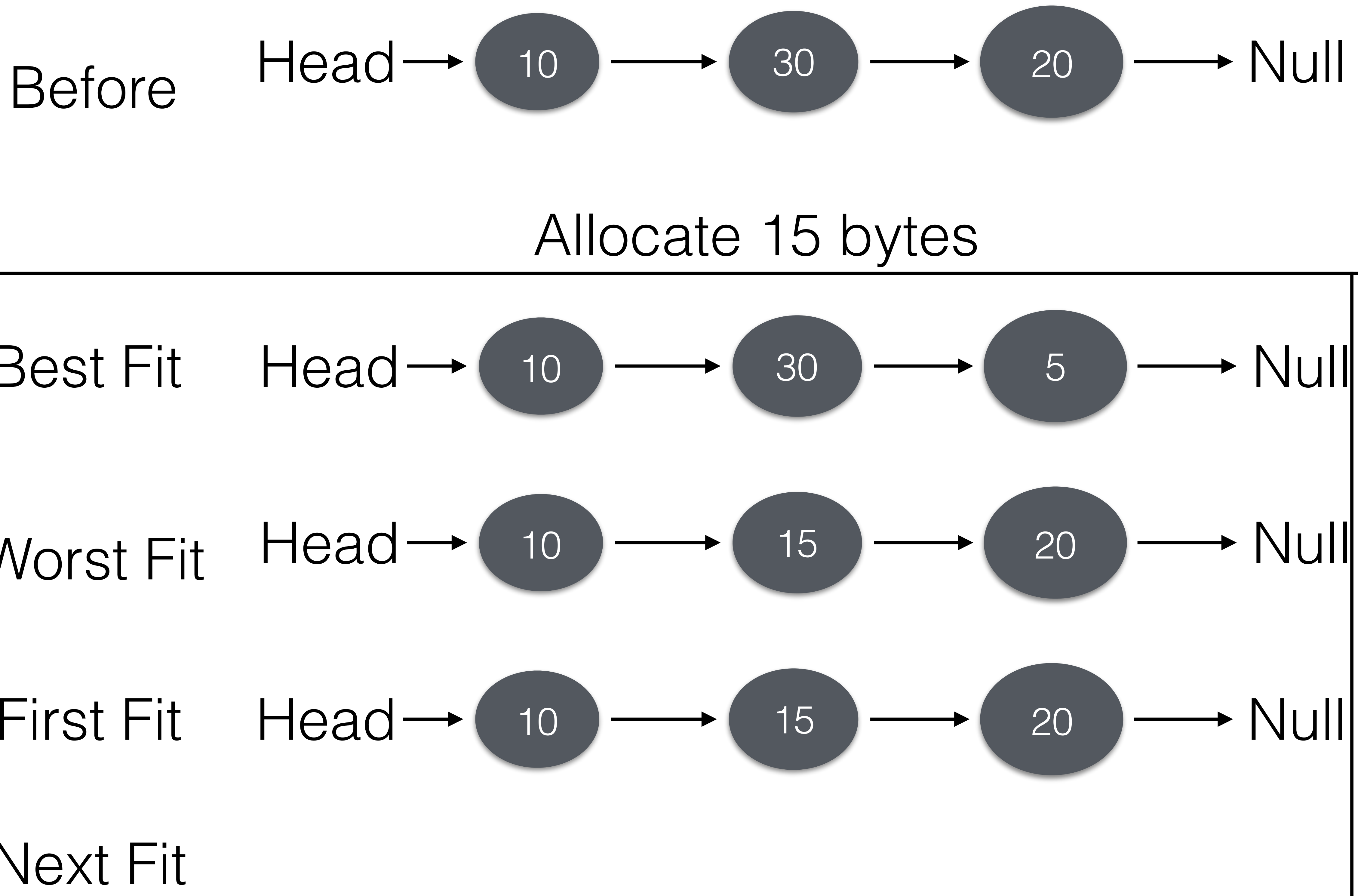
Allocation Strategies



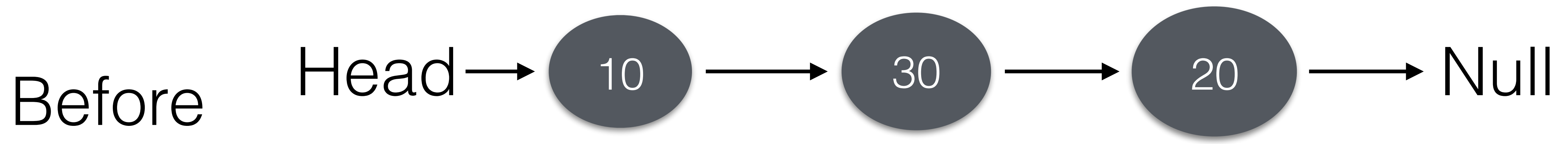
Allocation Strategies



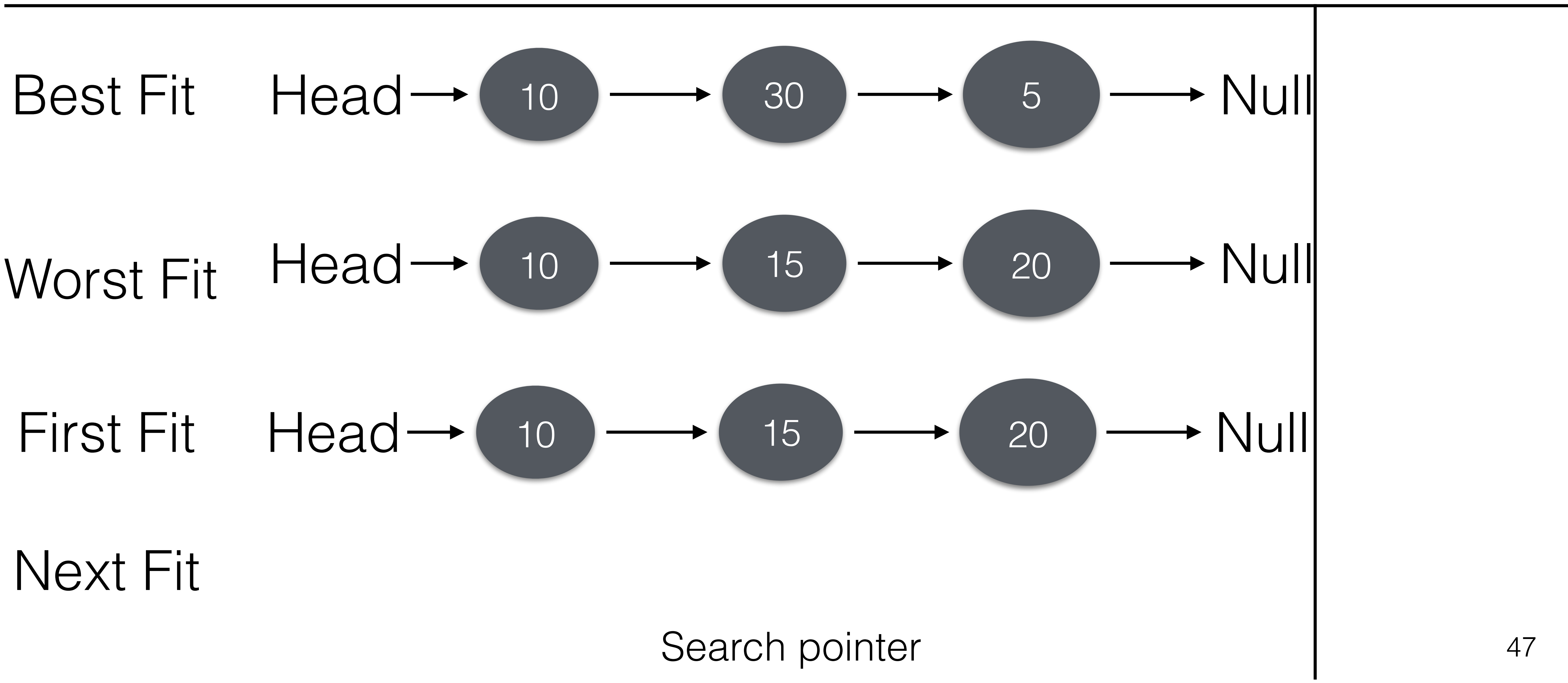
Allocation Strategies



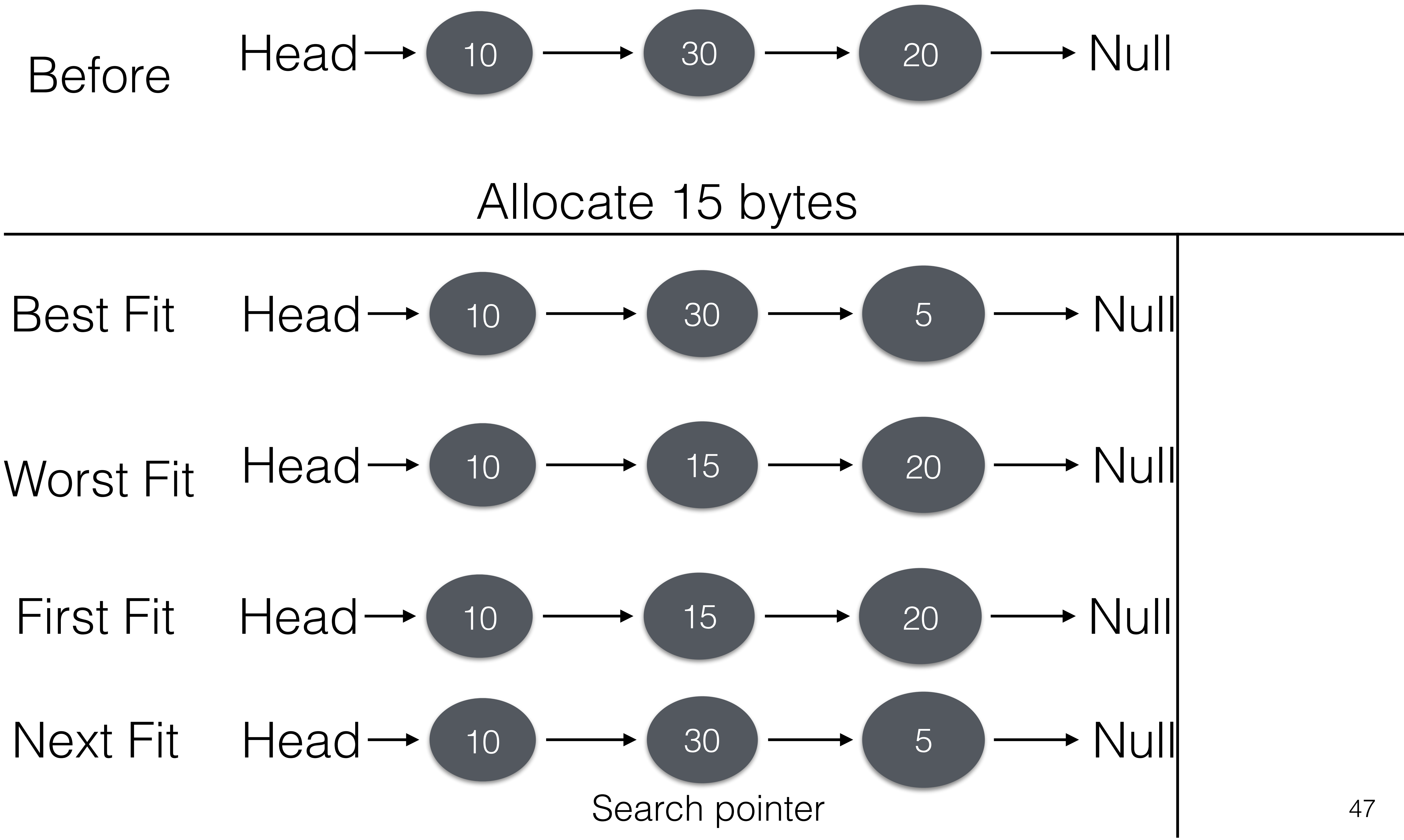
Allocation Strategies



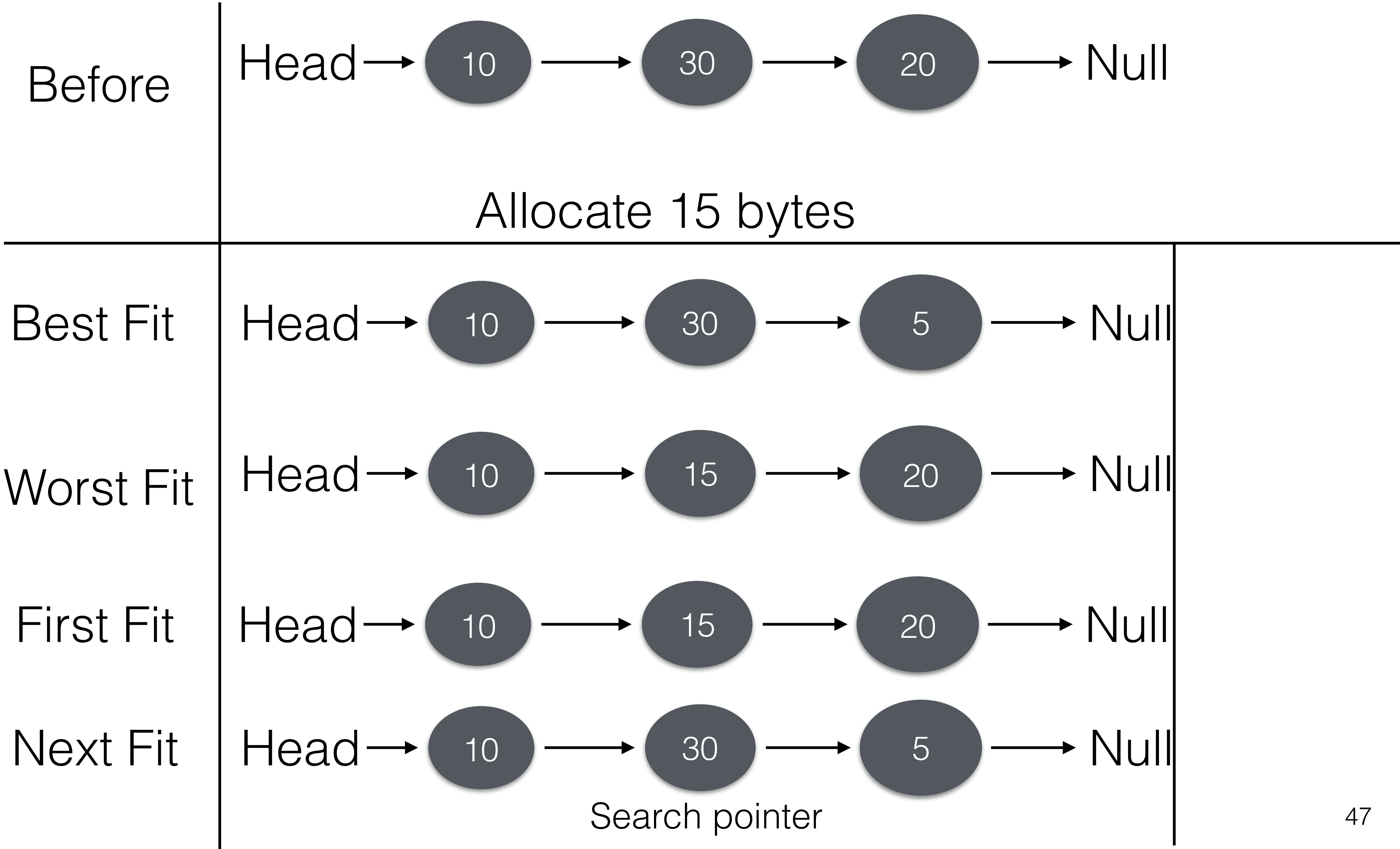
Allocate 15 bytes



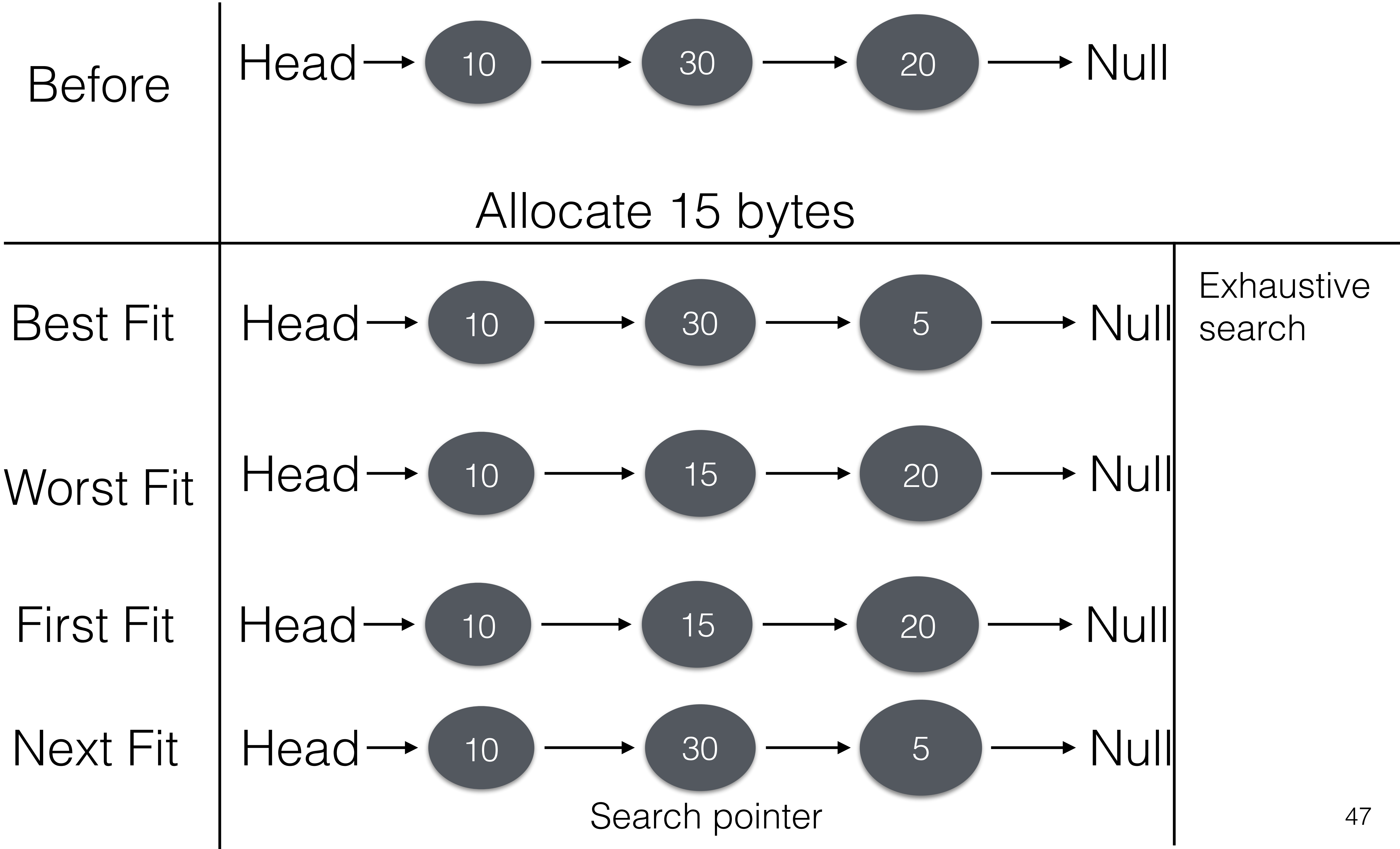
Allocation Strategies



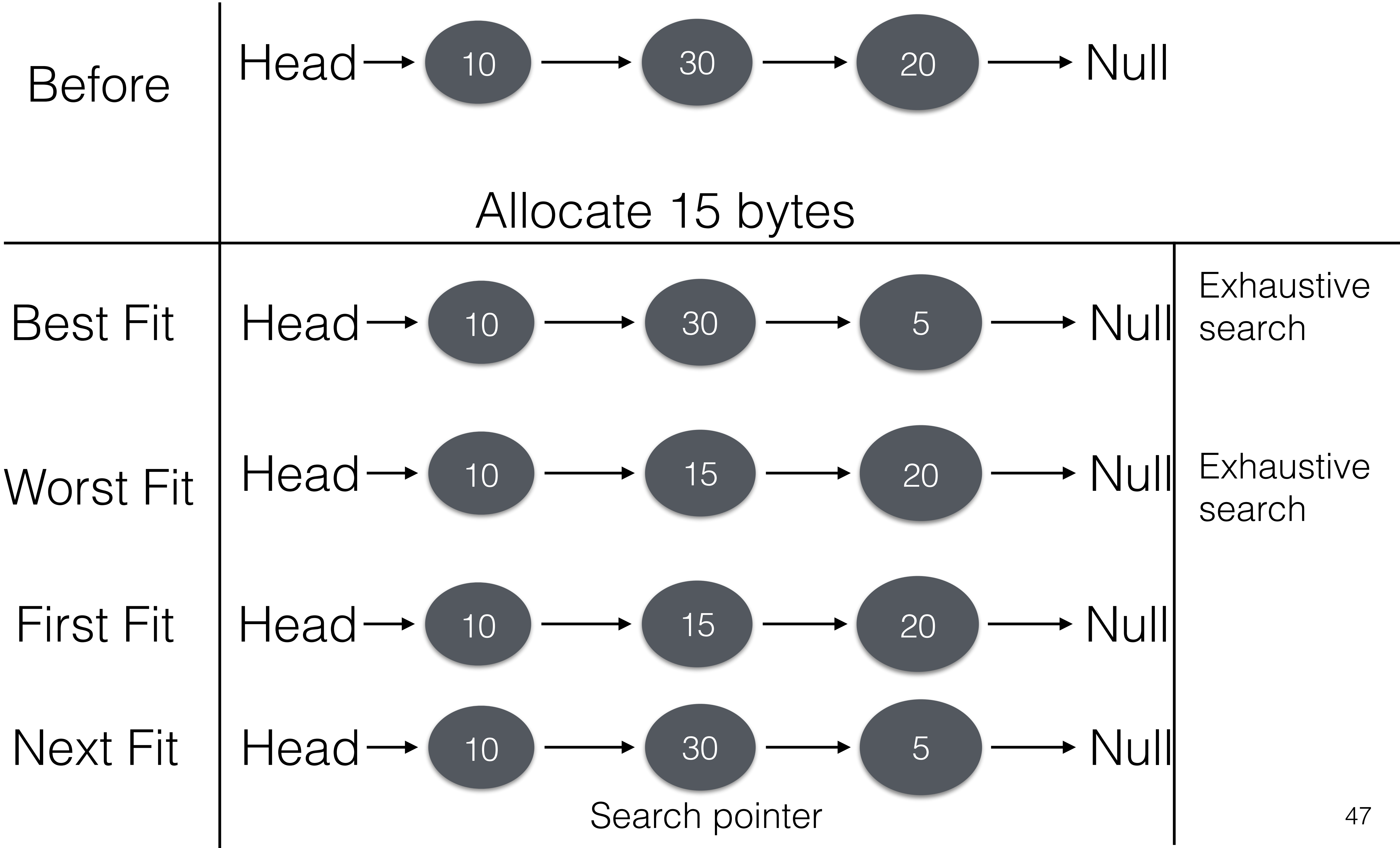
Allocation Strategies



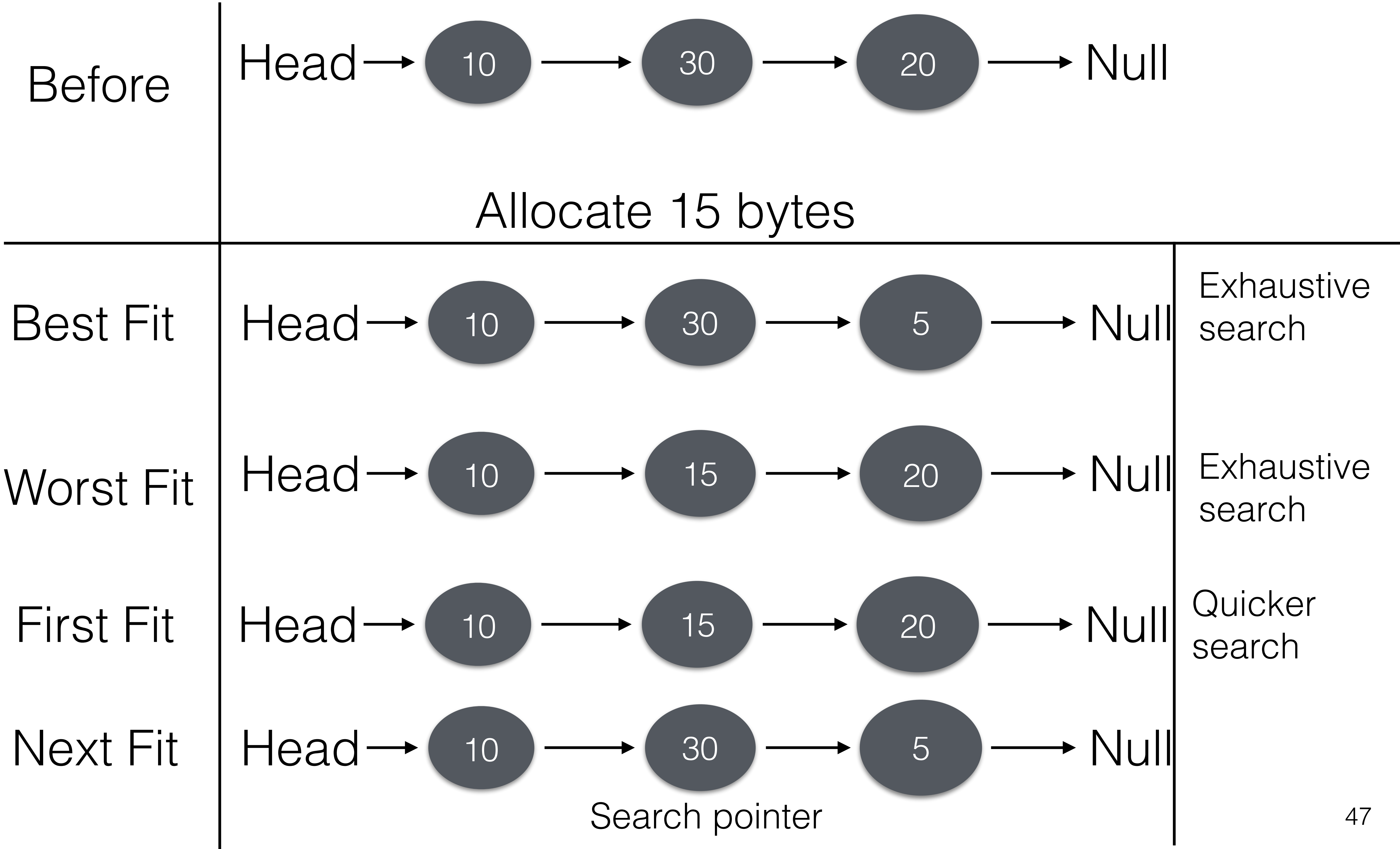
Allocation Strategies



Allocation Strategies


Before
Head → 10 → 30 → 20 → Null
Allocate 15 bytes
Best Fit
Head → 10 → 30 → 5 → Null
Exhaustive search
Worst Fit
Head → 10 → 15 → 20 → Null
Exhaustive search
First Fit
Head → 10 → 15 → 20 → Null
Next Fit
Head → 10 → 30 → 5 → Null
Search pointer

Allocation Strategies



Allocation Strategies

Before	Head → 10 → 30 → 20 → Null Allocate 15 bytes	
Best Fit	Head → 10 → 30 → 5 → Null	Exhaustive search
Worst Fit	Head → 10 → 15 → 20 → Null	Exhaustive search
First Fit	Head → 10 → 15 → 20 → Null	Quicker search
Next Fit	Head → 10 → 30 → 5 → Null Search pointer	Quicker search + Spread out

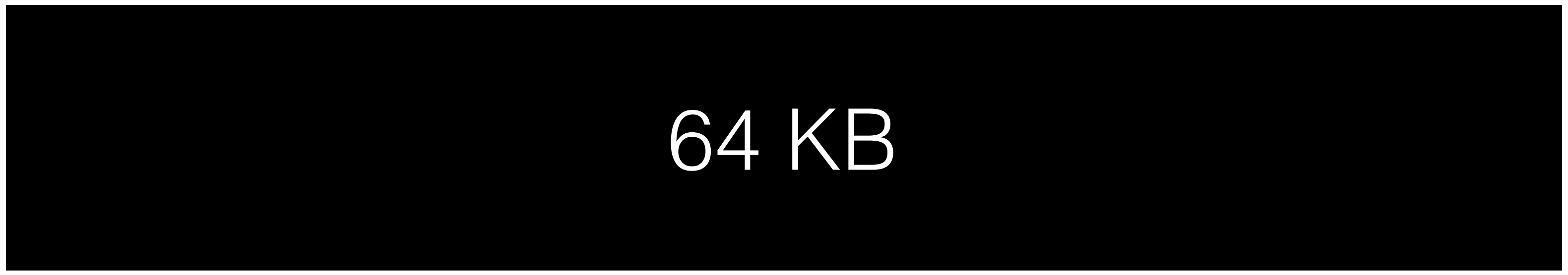
Binary Buddy Allocator

Binary Buddy Allocator

Assume free list = 2^N

Binary Buddy Allocator

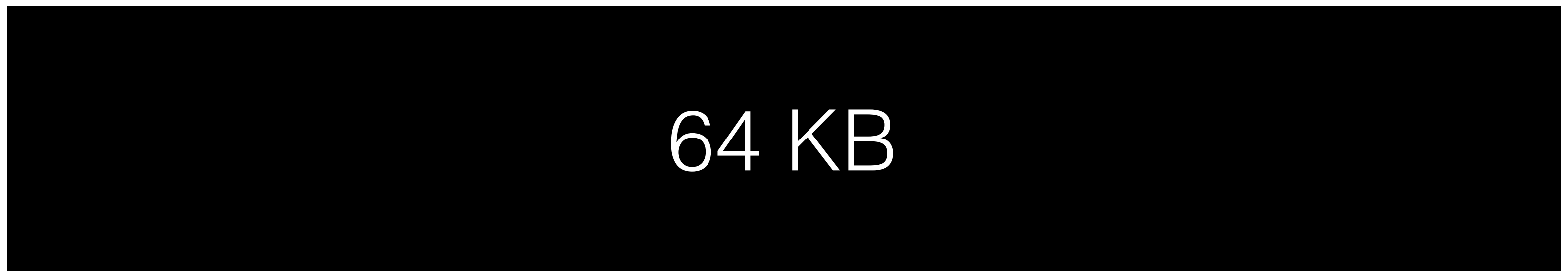
Assume free list = 2^N



Binary Buddy Allocator

Assume free list = 2^N

Want to allocate = 7 KB

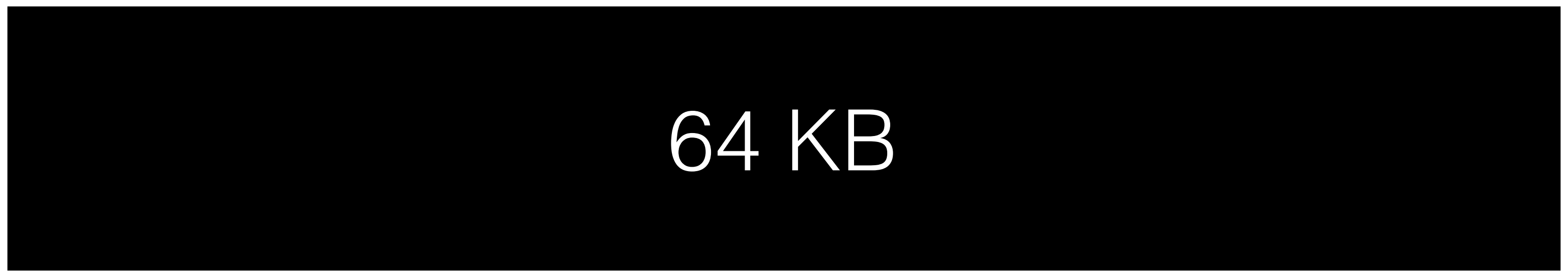


Binary Buddy Allocator

Assume free list = 2^N

Want to allocate = 7 KB

Recursively split till
we can do a “best fit”

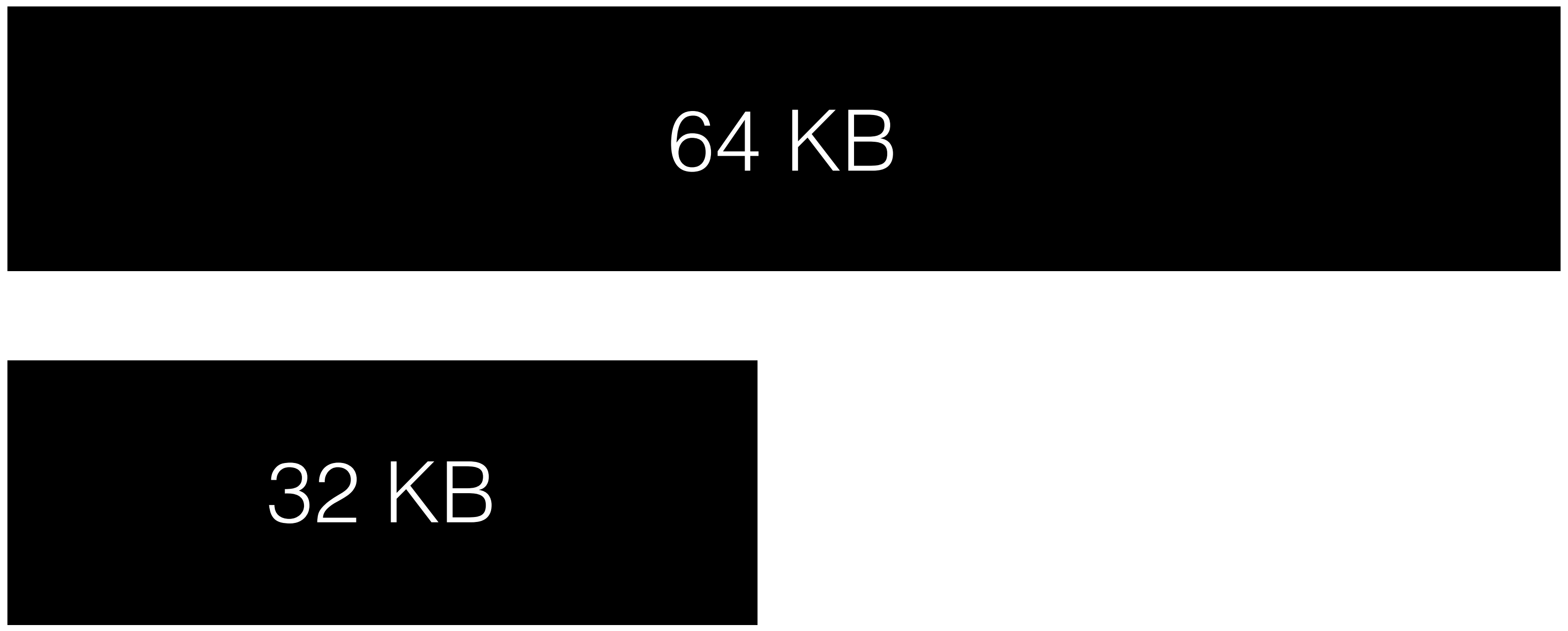


Binary Buddy Allocator

Assume free list = 2^N

Want to allocate = 7 KB

Recursively split till
we can do a "best fit"

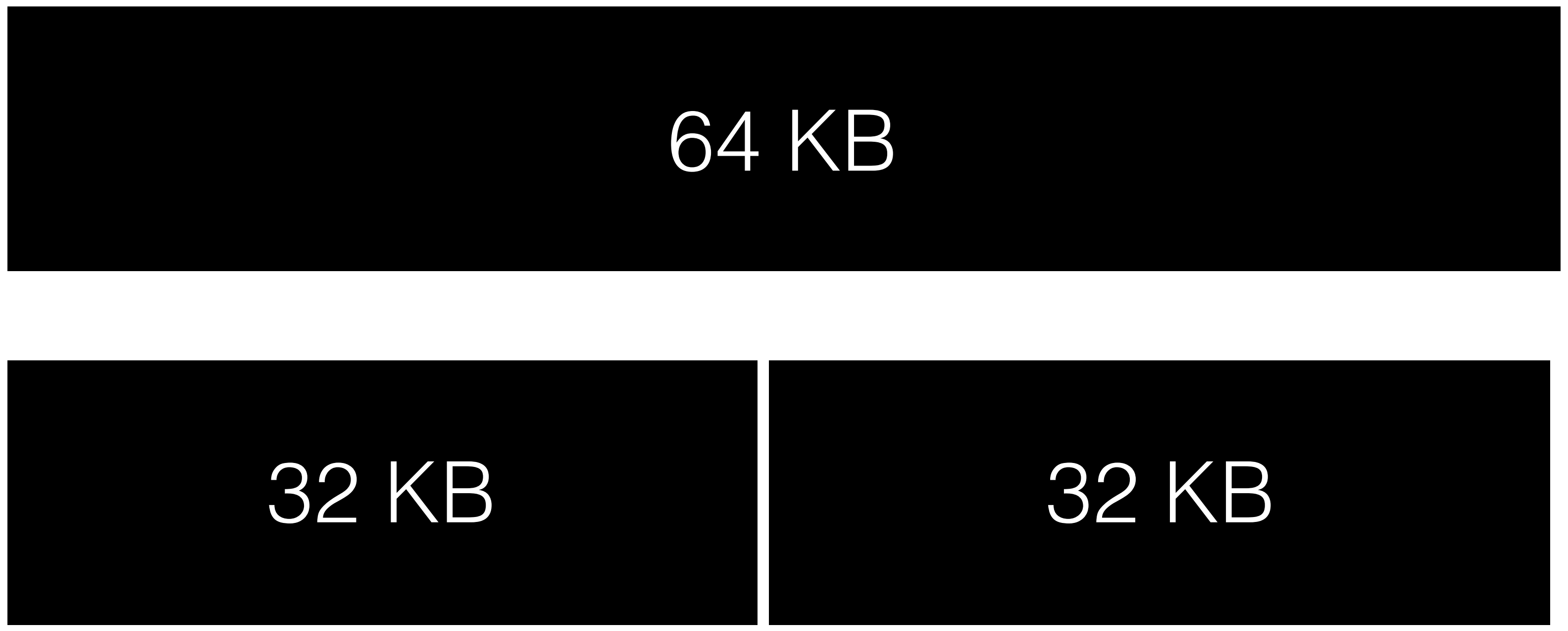


Binary Buddy Allocator

Assume free list = 2^N

Want to allocate = 7 KB

Recursively split till
we can do a "best fit"

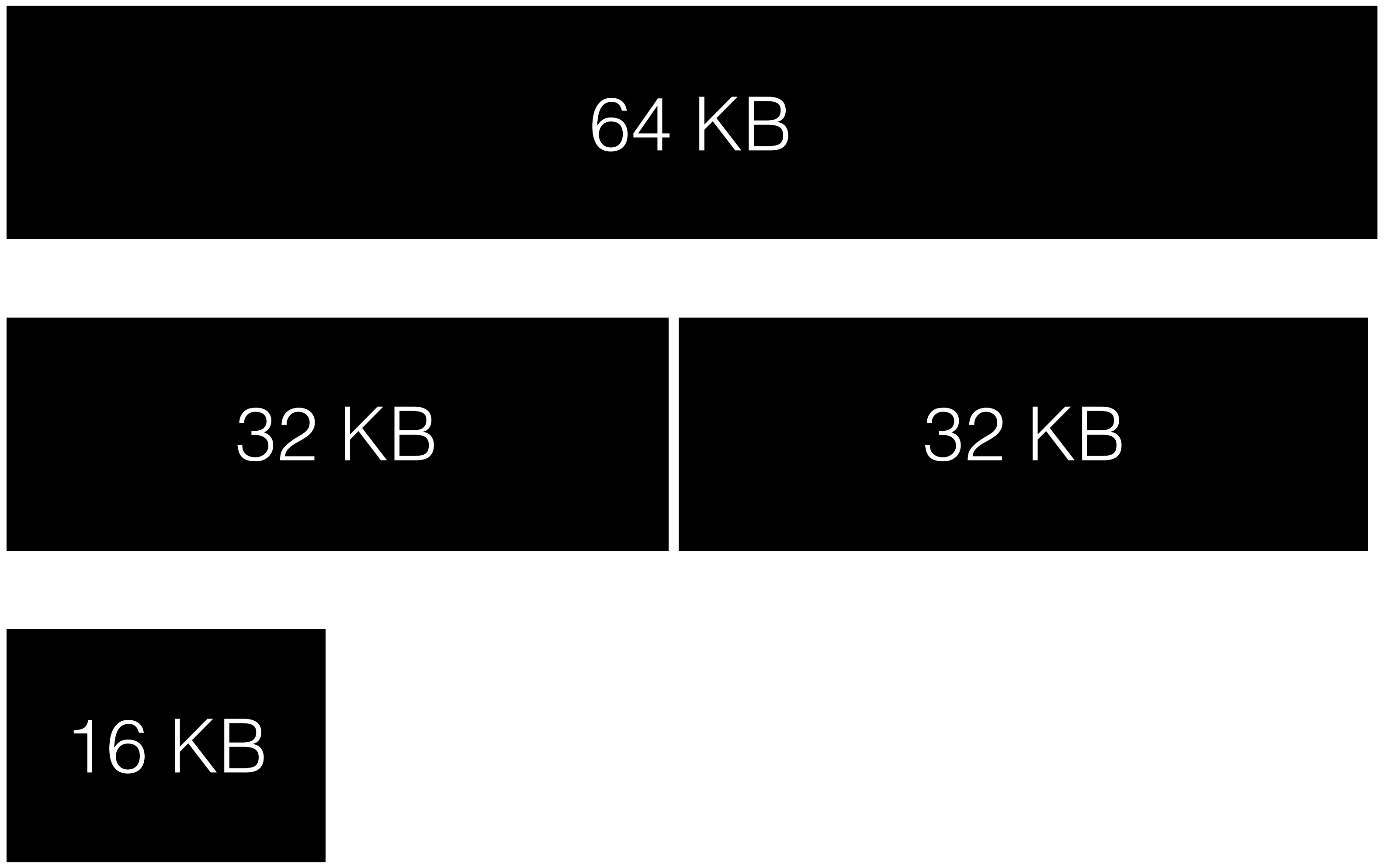


Binary Buddy Allocator

Assume free list = 2^N

Want to allocate = 7 KB

Recursively split till we can do a "best fit"

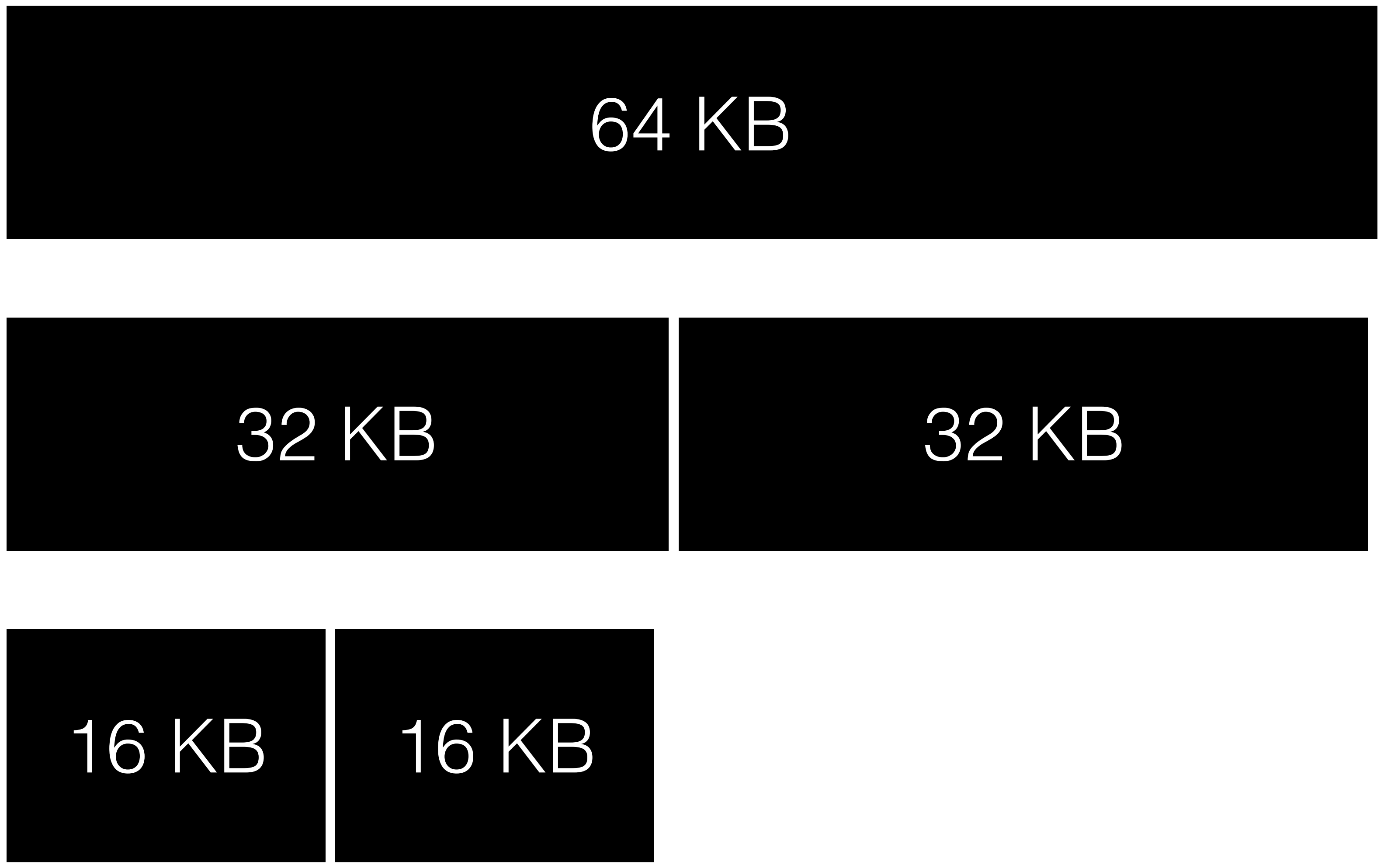


Binary Buddy Allocator

Assume free list = 2^N

Want to allocate = 7 KB

Recursively split till
we can do a “best fit”

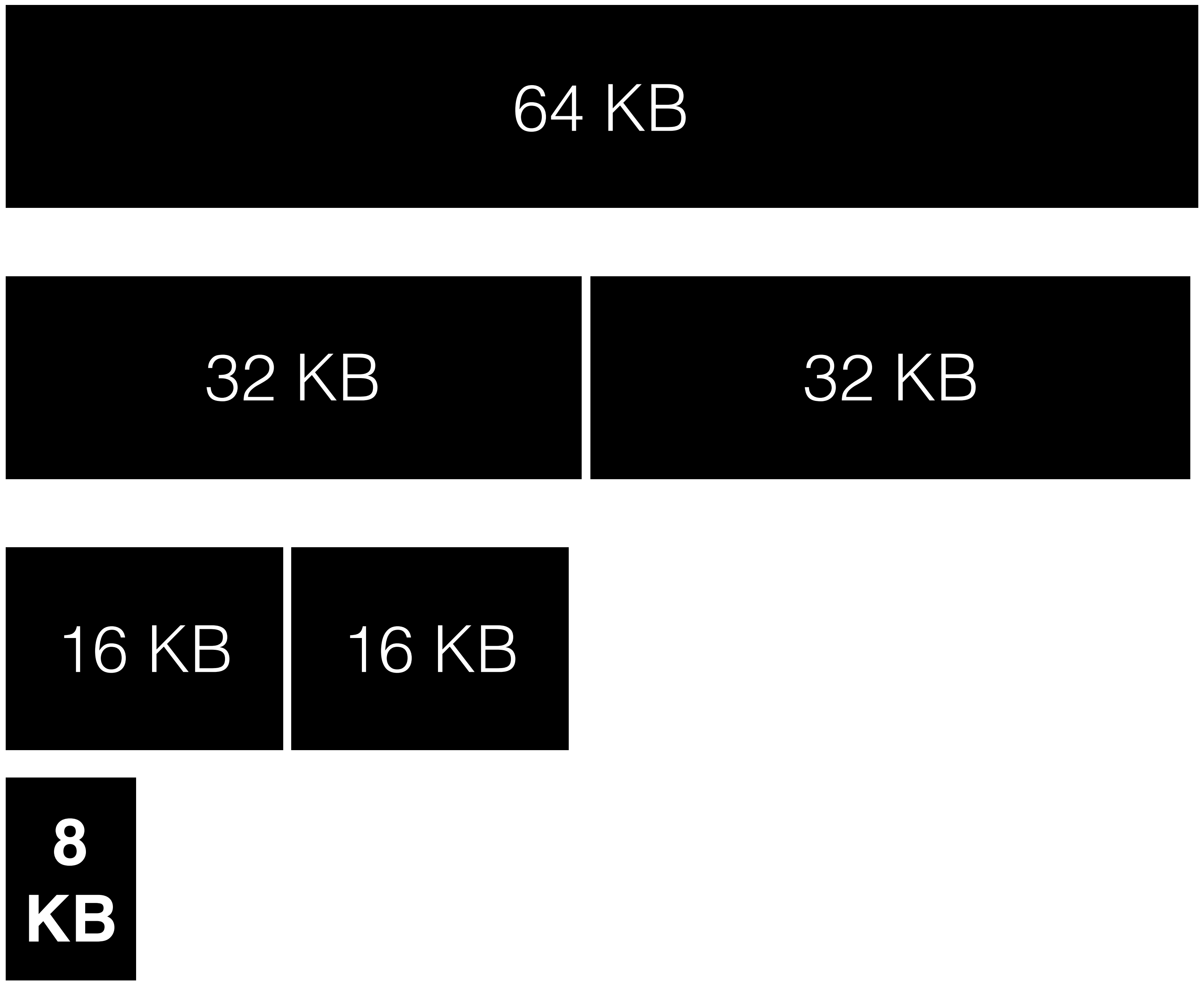


Binary Buddy Allocator

Assume free list = 2^N

Want to allocate = 7 KB

Recursively split till we can do a “best fit”

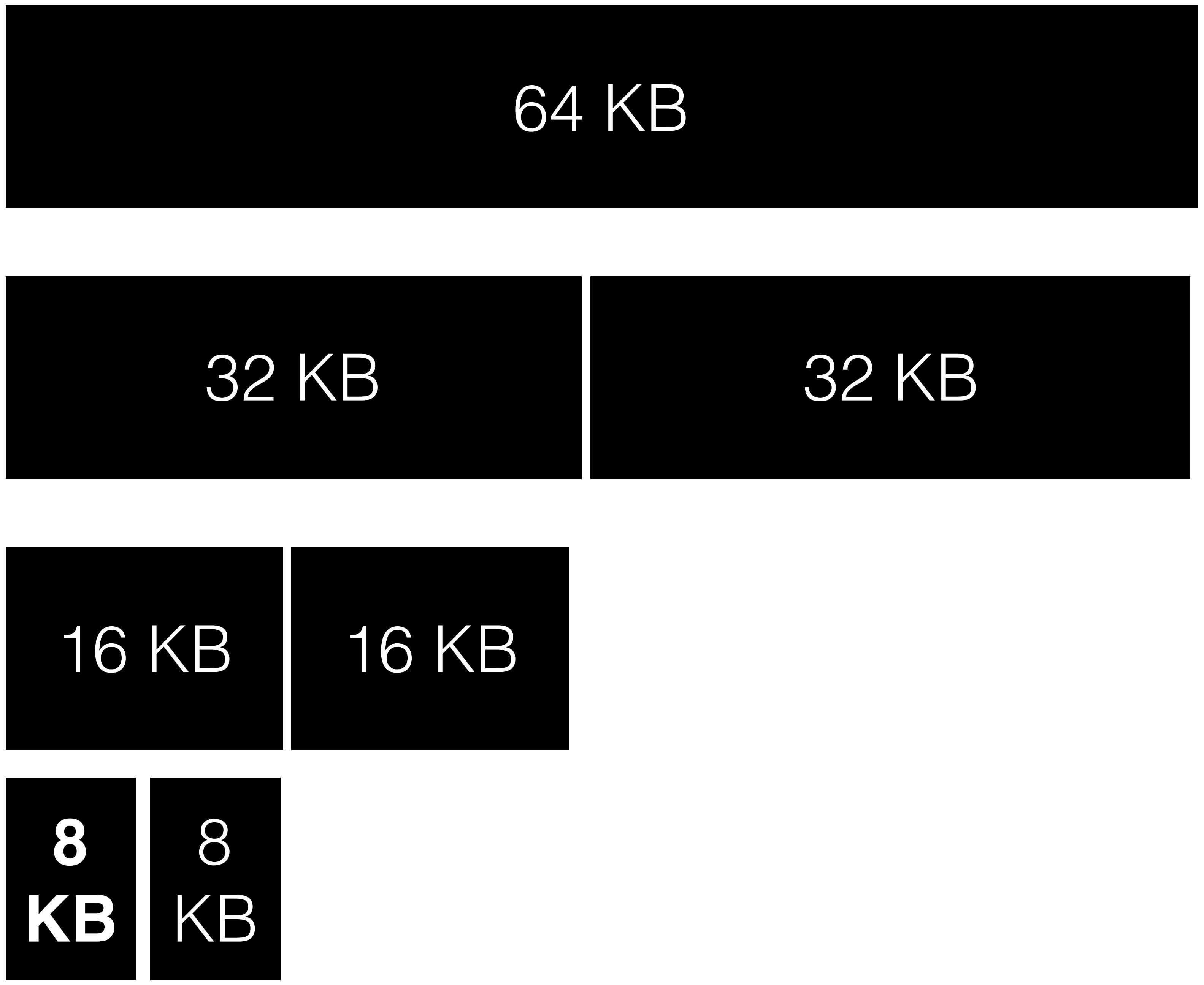


Binary Buddy Allocator

Assume free list = 2^N

Want to allocate = 7 KB

Recursively split till we can do a "best fit"



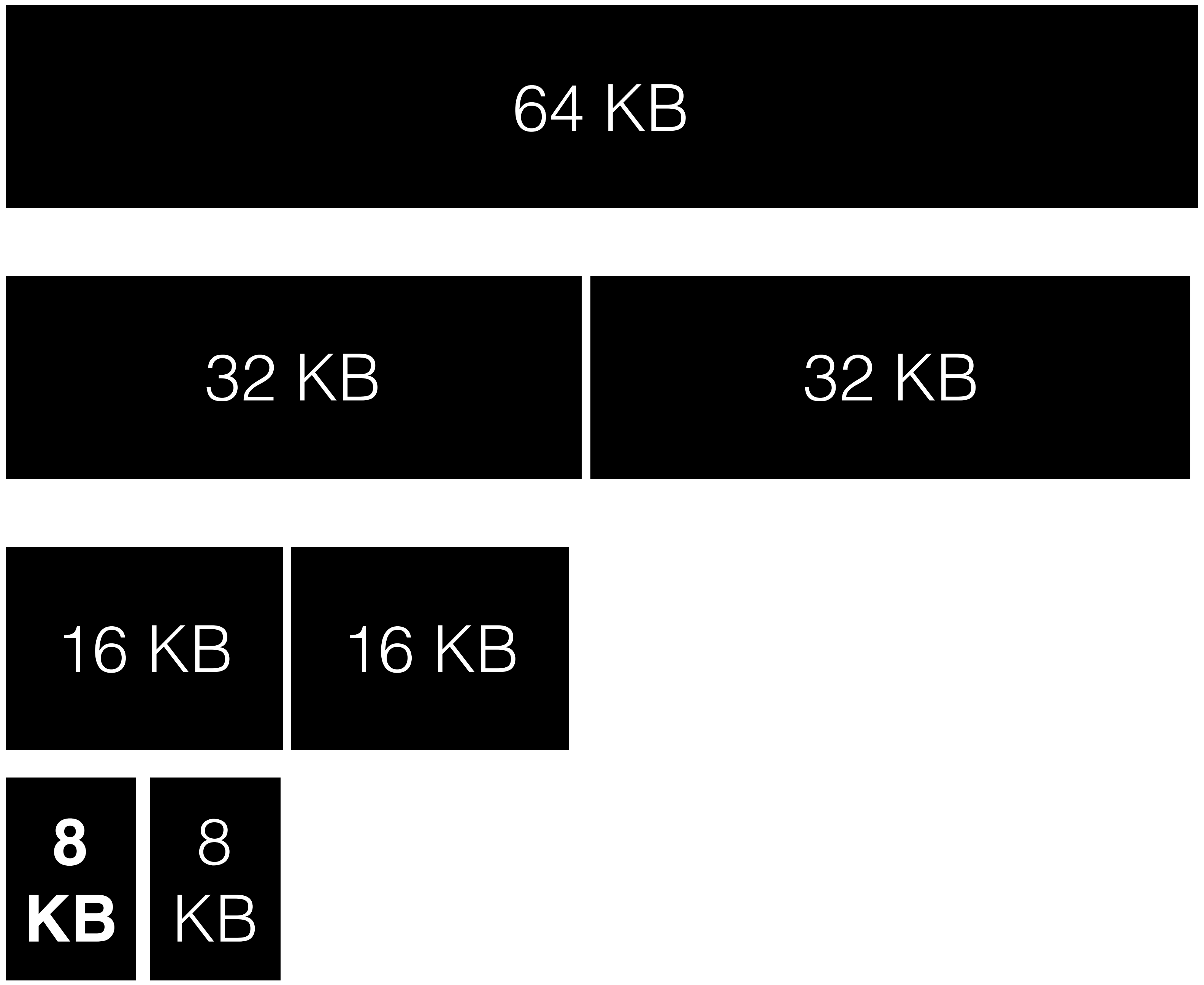
Binary Buddy Allocator

Assume free list = 2^N

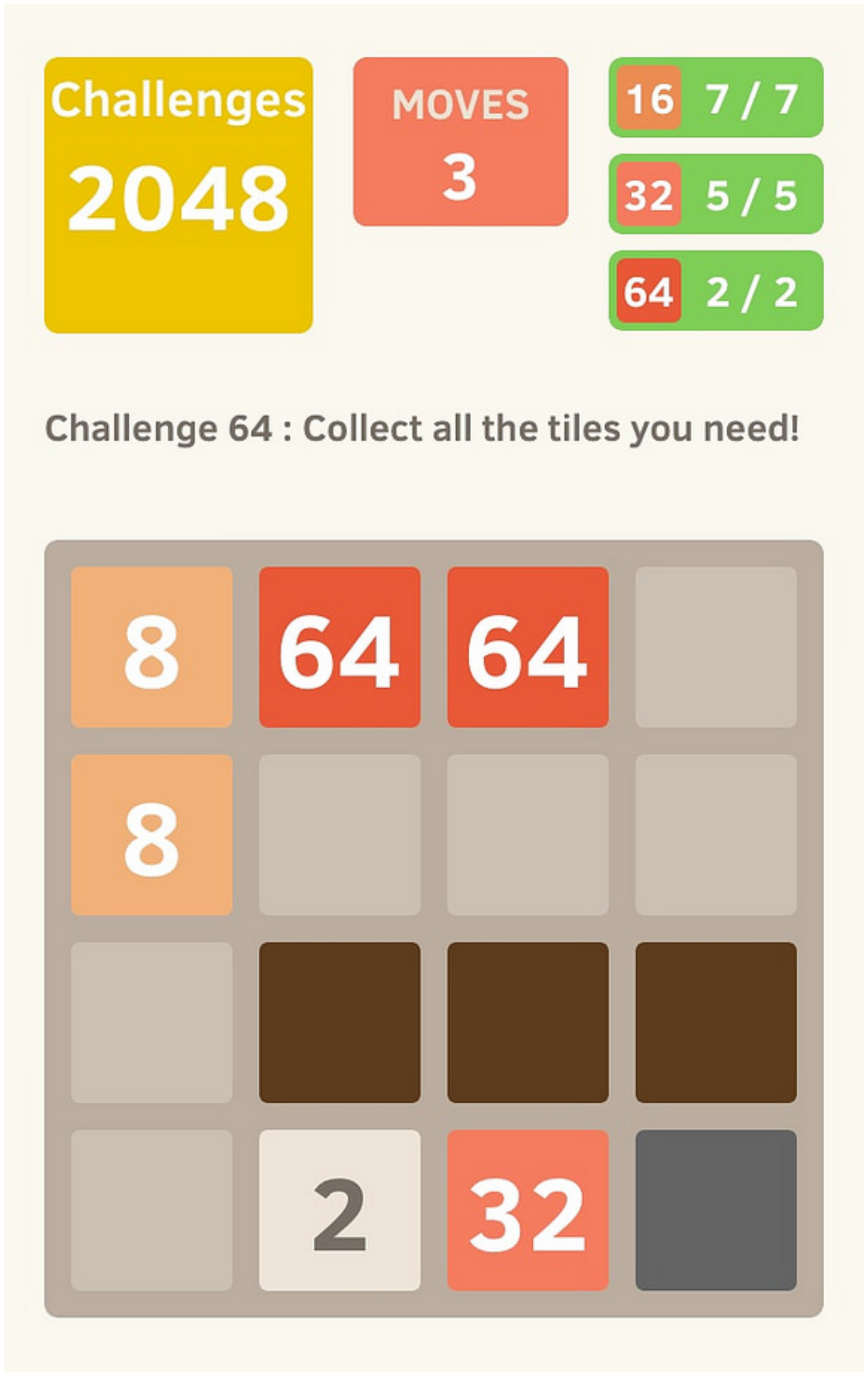
Want to allocate = 7 KB

Recursively split till
we can do a "best fit"

Allocate



Coalescing in Binary Buddy Algorithm



- Base & Bounds

- Base & Bounds
    - Pros: Very quick, 2 registers

- Base & Bounds
    - Pros: Very quick, 2 registers
    - Cons: Contiguous block of memory -> fragmentation

- Base & Bounds
  - Pros: Very quick, 2 registers
  - Cons: Contiguous block of memory -> fragmentation
- Segmentation

- Base & Bounds
  - Pros: Very quick, 2 registers
  - Cons: Contiguous block of memory -> fragmentation
- Segmentation
  - Pros: Still relatively simple, 3 registers, lesser fragmentation

- Base & Bounds
  - Pros: Very quick, 2 registers
  - Cons: Contiguous block of memory -> fragmentation
- Segmentation
  - Pros: Still relatively simple, 3 registers, lesser fragmentation
  - Cons: Still contiguous block of memory for segment

- Base & Bounds
    - Pros: Very quick, 2 registers
    - Cons: Contiguous block of memory -> fragmentation
- Segmentation
    - Pros: Still relatively simple, 3 registers, lesser fragmentation
    - Cons: Still contiguous block of memory for segment
- Paging

- Base & Bounds
  - Pros: Very quick, 2 registers
  - Cons: Contiguous block of memory -> fragmentation
- Segmentation
  - Pros: Still relatively simple, 3 registers, lesser fragmentation
  - Cons: Still contiguous block of memory for segment
- Paging
  - Pros: Very low chances of segmentation

- Base & Bounds
  - Pros: Very quick, 2 registers
  - Cons: Contiguous block of memory -> fragmentation
- Segmentation
  - Pros: Still relatively simple, 3 registers, lesser fragmentation
  - Cons: Still contiguous block of memory for segment
- Paging
  - Pros: Very low chances of segmentation
  - Cons: Slow, lots of memory accesses; memory overhead/process is huge!

- Base & Bounds
  - Pros: Very quick, 2 registers
  - Cons: Contiguous block of memory -> fragmentation
- Segmentation
  - Pros: Still relatively simple, 3 registers, lesser fragmentation
  - Cons: Still contiguous block of memory for segment
- Paging
  - Pros: Very low chances of segmentation
  - Cons: Slow, lots of memory accesses; memory overhead/process is huge!
- Paging + TLB

- Base & Bounds
  - Pros: Very quick, 2 registers
  - Cons: Contiguous block of memory -> fragmentation
- Segmentation
  - Pros: Still relatively simple, 3 registers, lesser fragmentation
  - Cons: Still contiguous block of memory for segment
- Paging
  - Pros: Very low chances of segmentation
  - Cons: Slow, lots of memory accesses; memory overhead/process is huge!
- Paging + TLB
  - Pros: Improves the address translation speed (spatial & temporal locality)

- Base & Bounds
  - Pros: Very quick, 2 registers
  - Cons: Contiguous block of memory -> fragmentation
- Segmentation
  - Pros: Still relatively simple, 3 registers, lesser fragmentation
  - Cons: Still contiguous block of memory for segment
- Paging
  - Pros: Very low chances of segmentation
  - Cons: Slow, lots of memory accesses; memory overhead/process is huge!
- Paging + TLB
  - Pros: Improves the address translation speed (spatial & temporal locality)
  - Cons: Limited in size, memory overhead/process still huge

- Base & Bounds
    - Pros: Very quick, 2 registers
    - Cons: Contiguous block of memory -> fragmentation
- Segmentation
    - Pros: Still relatively simple, 3 registers, lesser fragmentation
    - Cons: Still contiguous block of memory for segment
- Paging
    - Pros: Very low chances of segmentation
    - Cons: Slow, lots of memory accesses; memory overhead/process is huge!
- Paging + TLB
    - Pros: Improves the address translation speed (spatial & temporal locality)
    - Cons: Limited in size, memory overhead/process still huge
- Multi-level Paging

- Base & Bounds
    - Pros: Very quick, 2 registers
    - Cons: Contiguous block of memory -> fragmentation
- Segmentation
    - Pros: Still relatively simple, 3 registers, lesser fragmentation
    - Cons: Still contiguous block of memory for segment
- Paging
    - Pros: Very low chances of segmentation
    - Cons: Slow, lots of memory accesses; memory overhead/process is huge!
- Paging + TLB
    - Pros: Improves the address translation speed (spatial & temporal locality)
    - Cons: Limited in size, memory overhead/process still huge
- Multi-level Paging
    - Pros: Reduces memory overhead/process

- Base & Bounds
    - Pros: Very quick, 2 registers
    - Cons: Contiguous block of memory -> fragmentation
- Segmentation
    - Pros: Still relatively simple, 3 registers, lesser fragmentation
    - Cons: Still contiguous block of memory for segment
- Paging
    - Pros: Very low chances of segmentation
    - Cons: Slow, lots of memory accesses; memory overhead/process is huge!
- Paging + TLB
    - Pros: Improves the address translation speed (spatial & temporal locality)
    - Cons: Limited in size, memory overhead/process still huge
- Multi-level Paging
    - Pros: Reduces memory overhead/process
    - Cons: Cache miss can be very expensive!

- Base & Bounds
  - Pros: Very quick, 2 registers
  - Cons: Contiguous block of memory -> fragmentation
- Segmentation
  - Pros: Still relatively simple, 3 registers, lesser fragmentation
  - Cons: Still contiguous block of memory for segment
- Paging
  - Pros: Very low chances of segmentation
  - Cons: Slow, lots of memory accesses; memory overhead/process is huge!
- Paging + TLB
  - Pros: Improves the address translation speed (spatial & temporal locality)
  - Cons: Limited in size, memory overhead/process still huge
- Multi-level Paging
  - Pros: Reduces memory overhead/process
  - Cons: Cache miss can be very expensive!
- Swapping

- Base & Bounds
  - Pros: Very quick, 2 registers
  - Cons: Contiguous block of memory -> fragmentation
- Segmentation
  - Pros: Still relatively simple, 3 registers, lesser fragmentation
  - Cons: Still contiguous block of memory for segment
- Paging
  - Pros: Very low chances of segmentation
  - Cons: Slow, lots of memory accesses; memory overhead/process is huge!
- Paging + TLB
  - Pros: Improves the address translation speed (spatial & temporal locality)
  - Cons: Limited in size, memory overhead/process still huge
- Multi-level Paging
  - Pros: Reduces memory overhead/process
  - Cons: Cache miss can be very expensive!
- Swapping
  - Pros: Can transparently handle more memory than PA space

- Base & Bounds
  - Pros: Very quick, 2 registers
  - Cons: Contiguous block of memory -> fragmentation
- Segmentation
  - Pros: Still relatively simple, 3 registers, lesser fragmentation
  - Cons: Still contiguous block of memory for segment
- Paging
  - Pros: Very low chances of segmentation
  - Cons: Slow, lots of memory accesses; memory overhead/process is huge!
- Paging + TLB
  - Pros: Improves the address translation speed (spatial & temporal locality)
  - Cons: Limited in size, memory overhead/process still huge
- Multi-level Paging
  - Pros: Reduces memory overhead/process
  - Cons: Cache miss can be very expensive!
- Swapping
  - Pros: Can transparently handle more memory than PA space
  - Policies: to optimise what to keep in cache

- Base & Bounds
  - Pros: Very quick, 2 registers
  - Cons: Contiguous block of memory -> fragmentation
- Segmentation
  - Pros: Still relatively simple, 3 registers, lesser fragmentation
  - Cons: Still contiguous block of memory for segment
- Paging
  - Pros: Very low chances of segmentation
  - Cons: Slow, lots of memory accesses; memory overhead/process is huge!
- Paging + TLB
  - Pros: Improves the address translation speed (spatial & temporal locality)
  - Cons: Limited in size, memory overhead/process still huge
- Multi-level Paging
  - Pros: Reduces memory overhead/process
  - Cons: Cache miss can be very expensive!
- Swapping
  - Pros: Can transparently handle more memory than PA space
  - Policies: to optimise what to keep in cache
- Free space managament

Next time —> Memory Virtualisation in Linux