Operating Systems COMS(3010A) Memory

Branden Ingram
branden.ingram@wits.ac.za
Office Number : ???

Recap

- Interrupt Stack
- System Calls
- UpCalls
- UNIX

Memory Virtualization

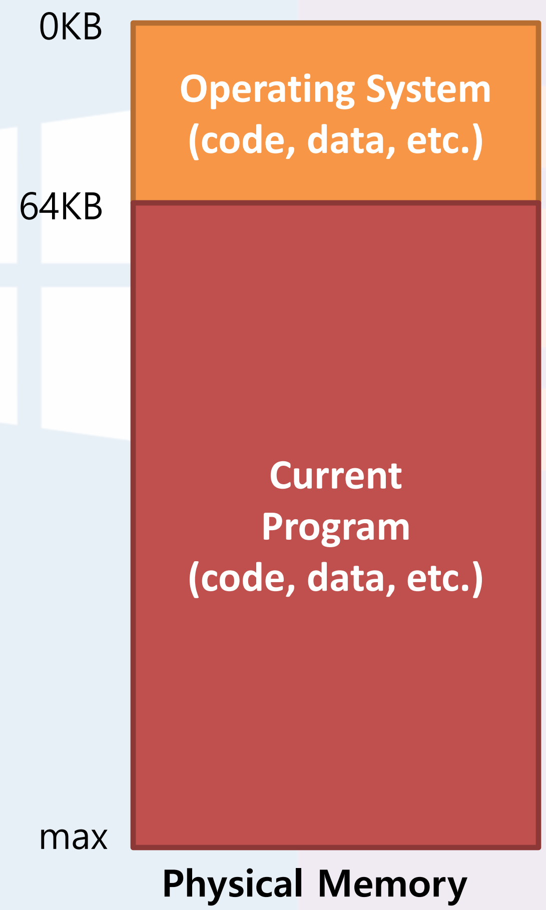
- **What is memory virtualization?**
  - OS virtualizes its physical memory.
  - OS provides an illusion of memory space per each process
  - It seems to be seen like each process uses the whole memory

What are the benefits of Memory Virtualization

- **Ease of use in programming**
- **Memory efficiency in terms of times and space**
- **The guarantee of isolation for processes as well as OS**
    - Protection from errant accesses of other processes

Memory Management in Early OS's

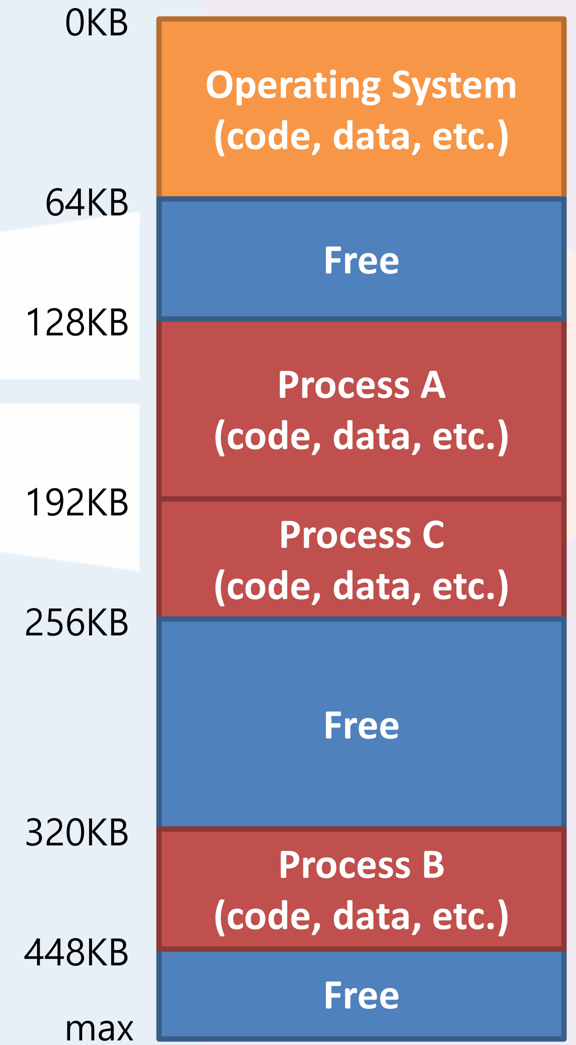
- **Load only one process in memory**
  - Poor utilization and efficiency



Multiprogramming and Time Sharing

- **Load multiple processes in memory**
  - Execute one for a short while
  - Switch processes between them in memory
  - Increase utilization and efficiency

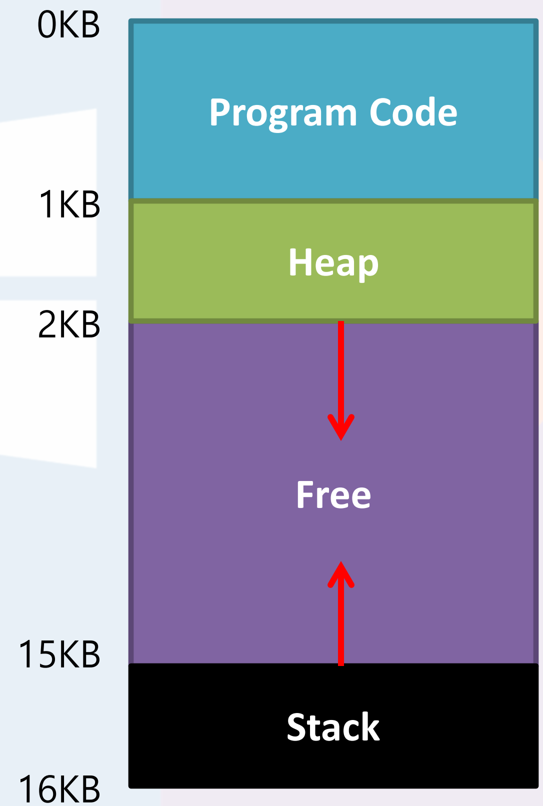
- **Cause an important protection issue**
  - Errant memory accesses from other processes



Physical Memory

Address Space

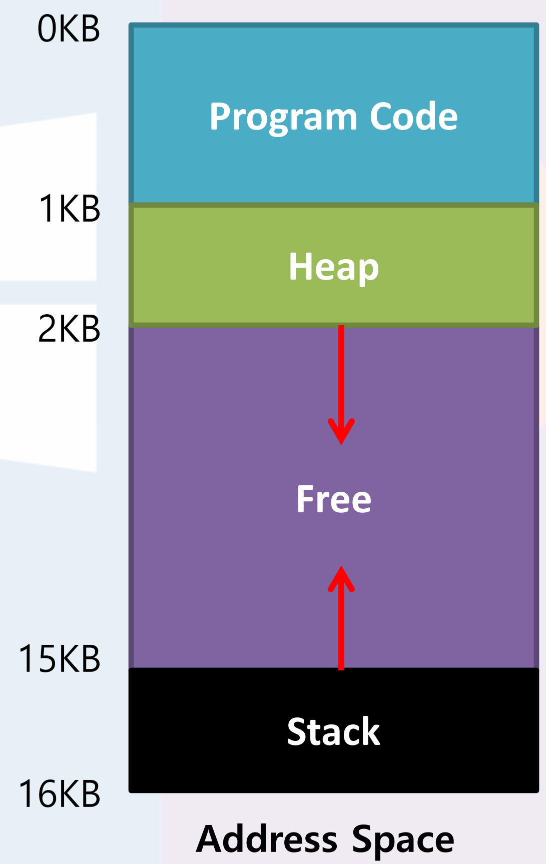
- **OS creates an abstraction of physical memory**
  - This is called the Address Space
  - The address space contains all the info about a running process
  - That is it consists of program code, heap, stack and etc



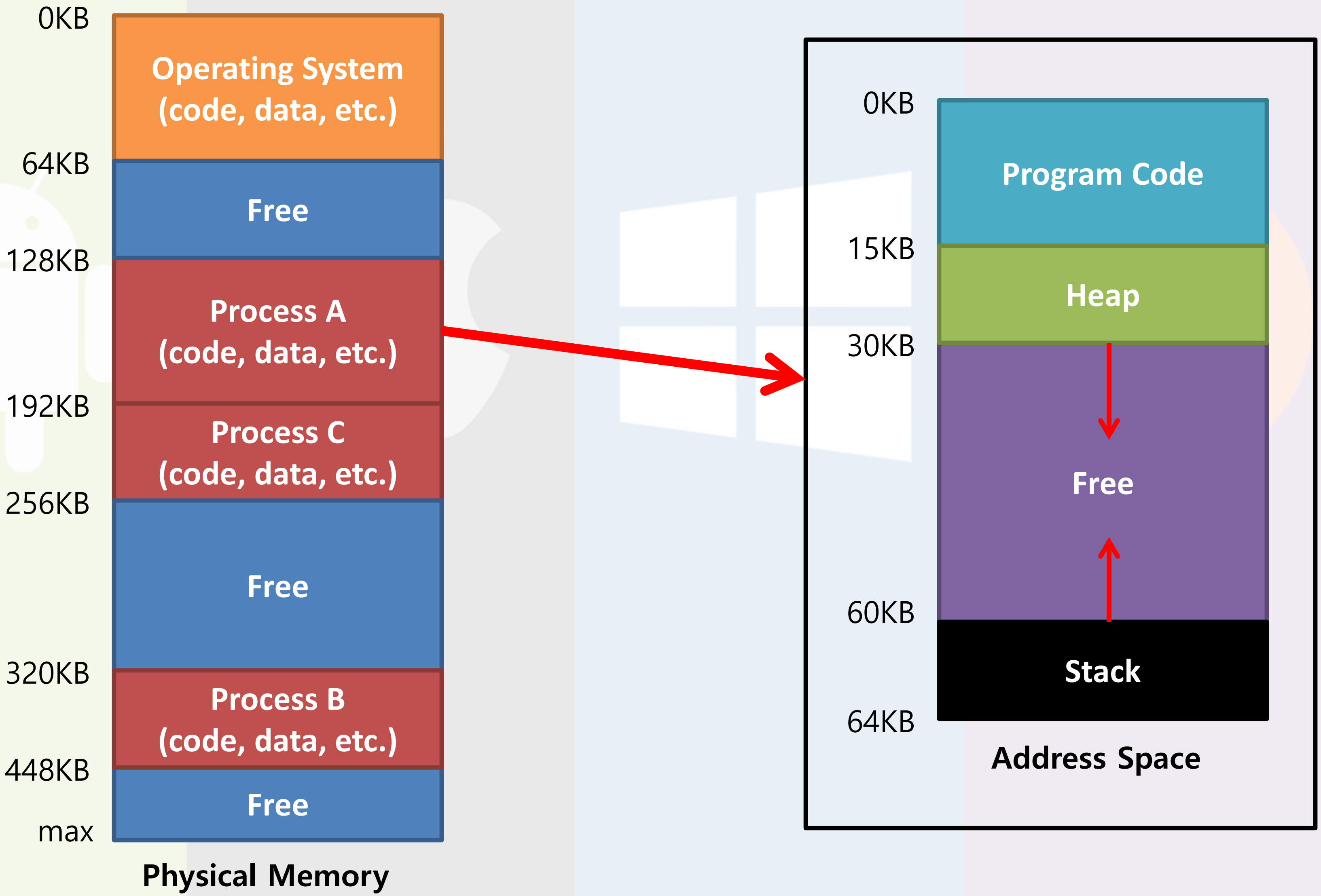
Address Space

Address Space

- **Code**
  - Where instructions live
- **Heap**
  - Dynamically allocate memory
    - “malloc” in C language
    - “new” in object-oriented language
- **Stack**
  - Store return addresses or values
  - Contain local variables arguments to routines



Address Space



Well how do we map between Physical Memory and the Address Space?

- Well the OS ofcourse

Well how do we map between Physical Memory and the Address Space?

- **Well the OS of course**

- **Every address in a running program is virtual**
  - This “Virtual Address” is the address in the Address Space
- **OS translates the virtual address to physical address**

Virtual Addresses

- The virtualized address in address space

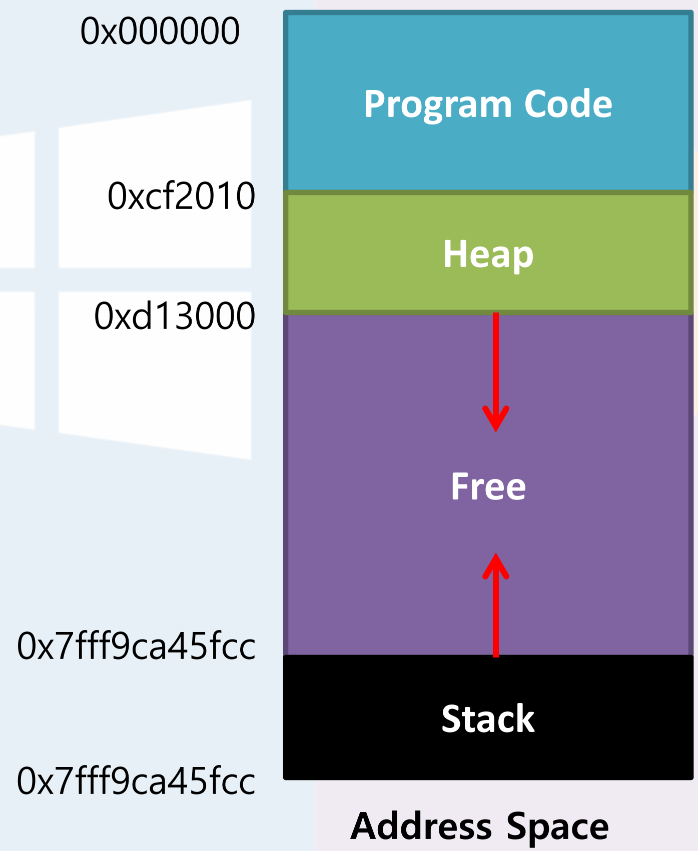
```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]){

    printf("location of code  : %p\n", (void *) main);
    printf("location of heap  : %p\n", (void *) malloc(1));
    int x = 3;
    printf("location of stack : %p\n", (void *) &x);

    return x;
}
```

A simple program that prints out addresses

Virtual Addresses

- **The output in 64-bit Linux machine**

```
location of code  : 0x000000
location of heap  : 0xcf2010
location of stack : 0x7fff9ca45fcc
```



Address Translation

- **Hardware transforms a virtual address to a physical address**
  - The desired information is actually stored in a physical address

- **The OS must get involved at key points to set up the hardware**
  - The OS must manage memory to judiciously intervene

Address Translation - Example

- **C - Language code**

```
void func()
        int x;
        ...
        x = x + 3; // this is the line of code we are interested in
```

  - Load a value from memory
  - Increment it by three
  - Store the value back into memory

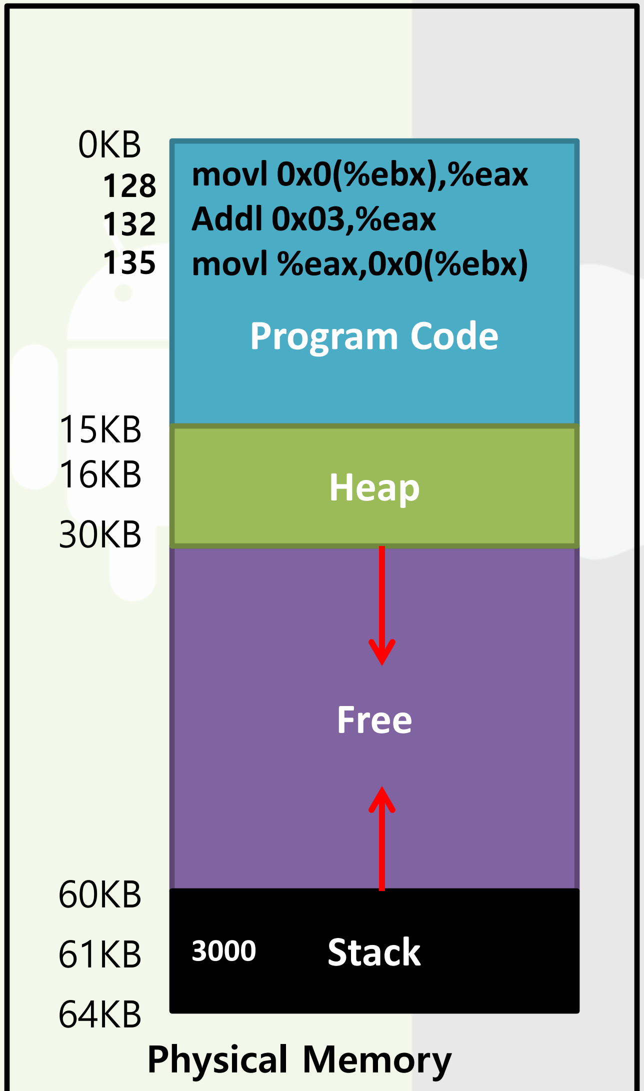
Address Translation - Example

- **Assembly**

```
128 : movl 0x0(%ebx), %eax        ; load 0+ebx into eax
132 : addl $0x03, %eax            ; add 3 to eax register
135 : movl %eax, 0x0(%ebx)        ; store eax back to mem
```

- Presume that the address of ‘x’ has been place in ebx register
- Load the value at that address into eax register
- Add 3 to eax register
- Store the value in eax back into memory

Address Translation - Example

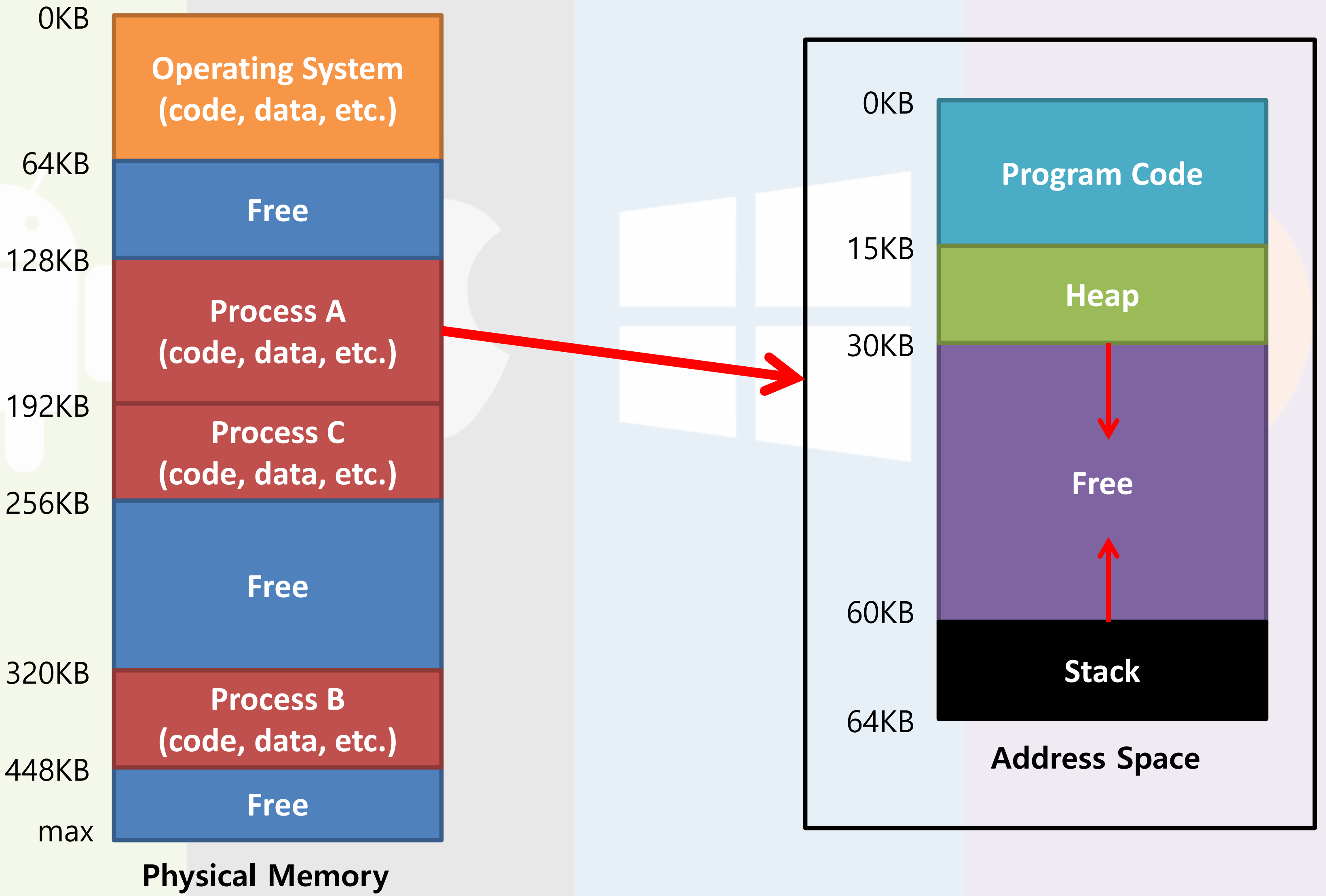


- Fetch instruction at address 128
- Execute this instruction (load from address 61KB)
- Fetch instruction at address 132
- Execute this instruction (no memory reference)
- Fetch the instruction at address 135
- Execute this instruction (store to address 61 KB)

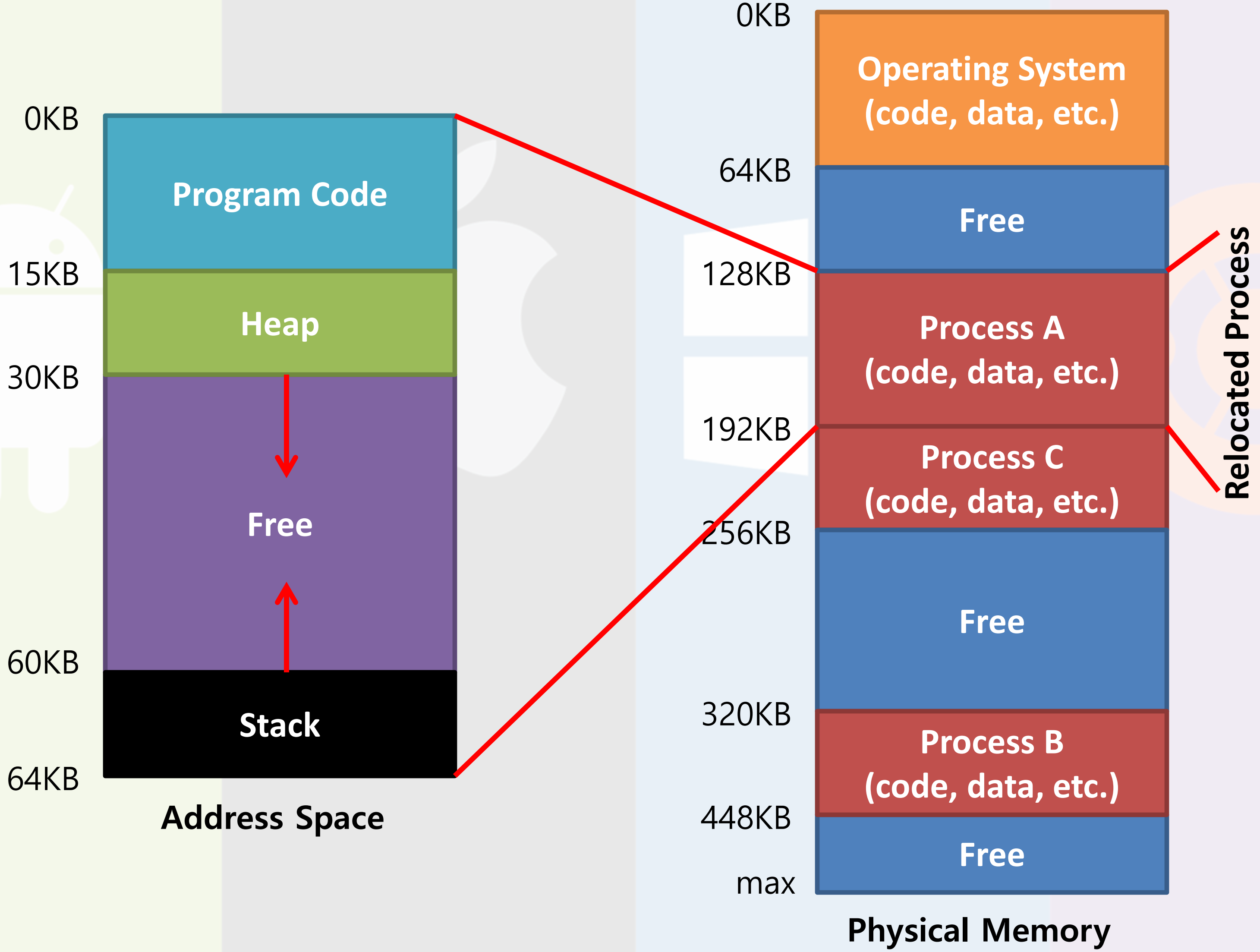
What happens if the OS needs to relocate a process?

- **The OS wants to place the process somewhere else in physical memory, not at address 0**
    - The address space start at address 0

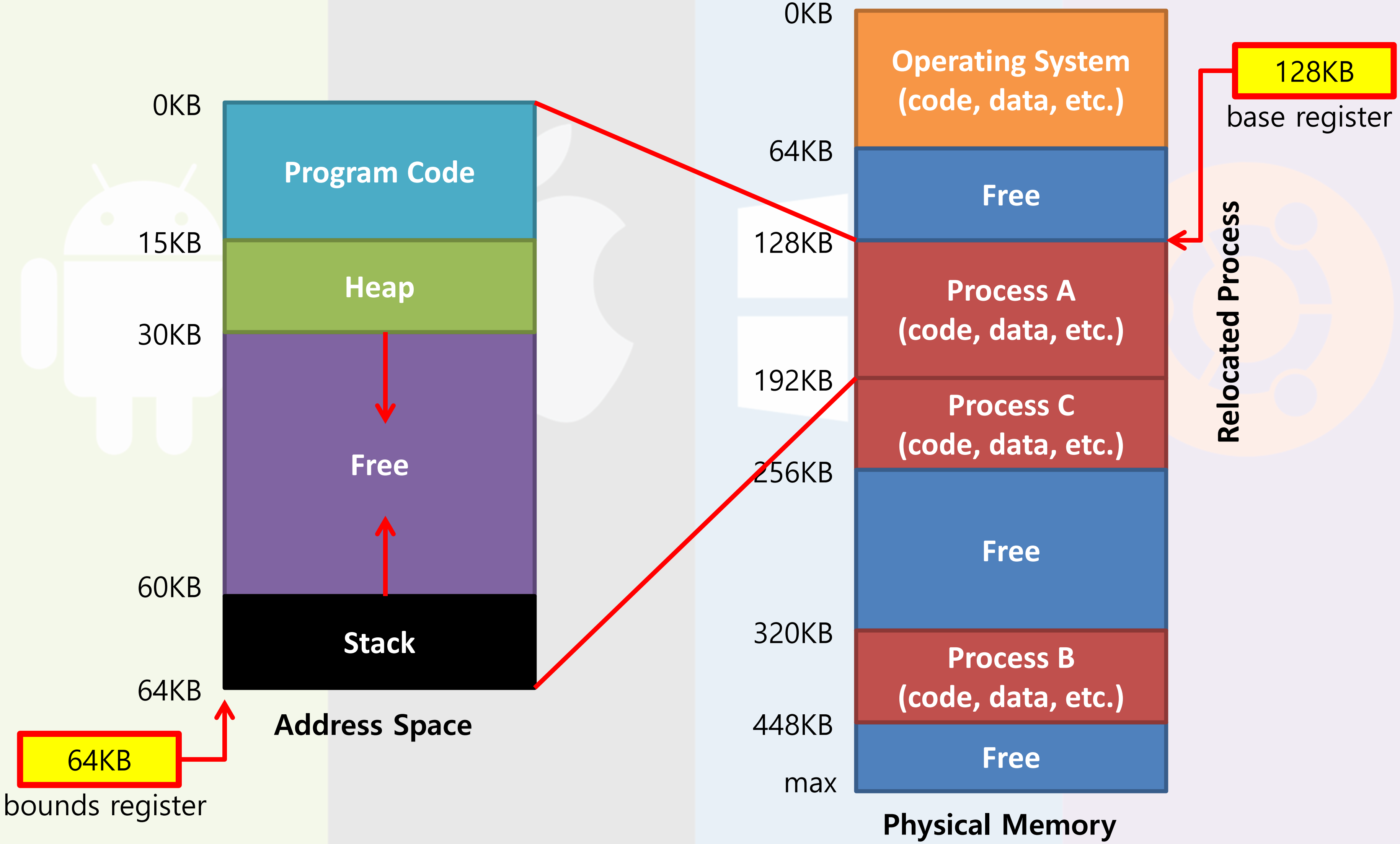
Memory Relocation



Memory Relocation



Base and Bounds Registers



Hardware base Relocation

- **When a program starts running, the OS decides where in physical memory a process should be loaded**
  - Set the **base** register value

Physical address = virtual address + base

  - Every virtual address must **not be greater than bound** and **negative**

0 <= virtual address < bounds

Address Translation

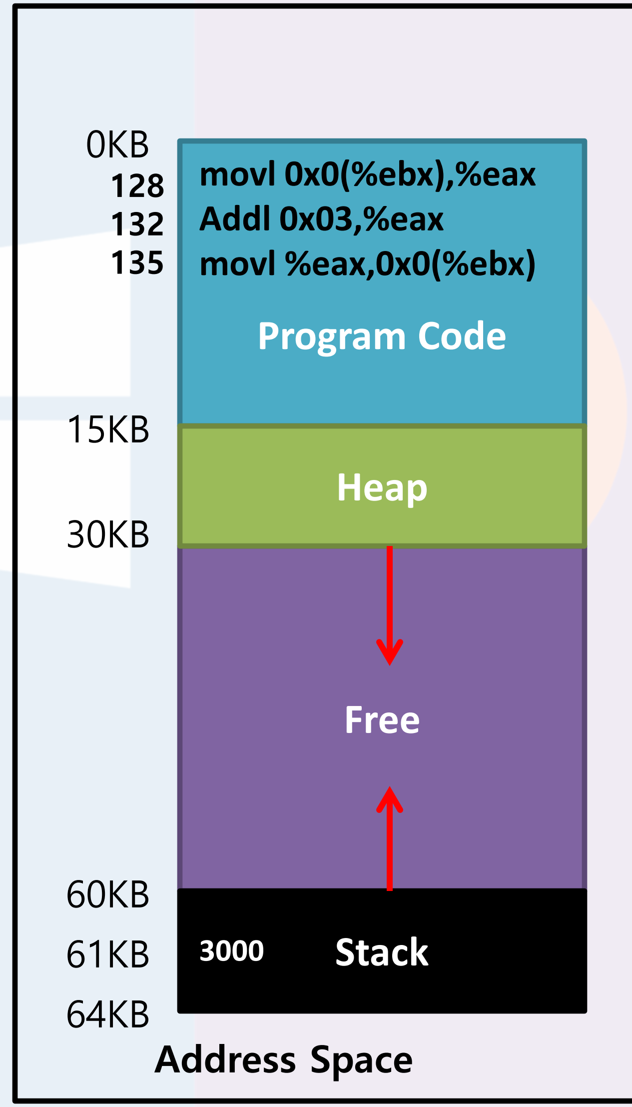
```
128 : movl 0x0(%ebx), %eax
```

- **Fetch instruction at address 128**

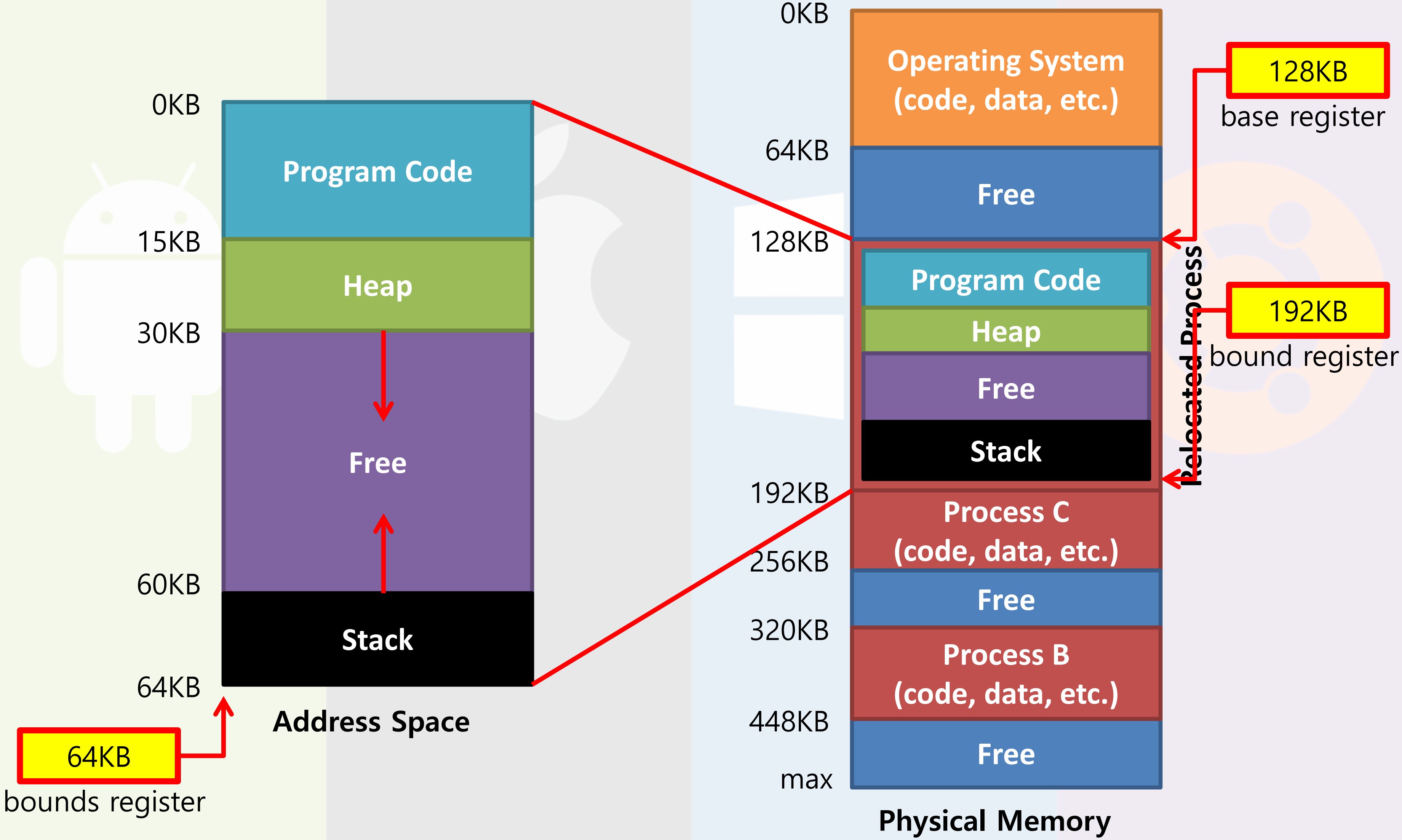
131200 = 128 + 128KB(base)

- **Execute this instruction**
  - Load from address 61KB

189KB = 61KB + 128KB(base)



Two ways of Bounds Register



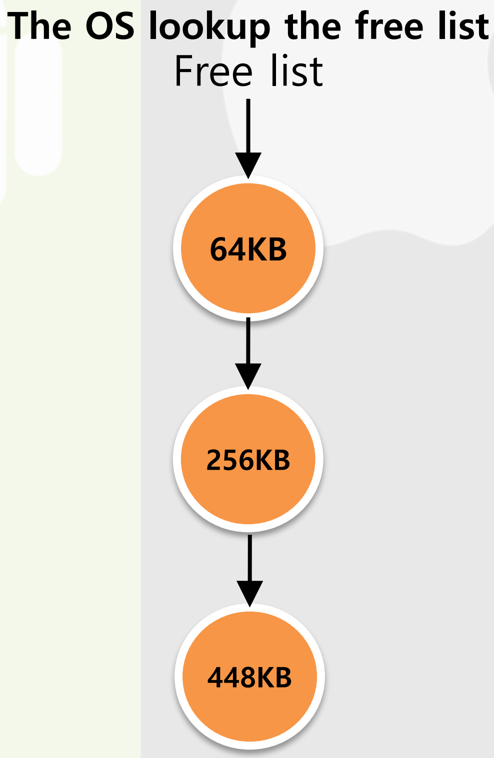
What are the Issues an OS must handle for Memory Virtualizing?

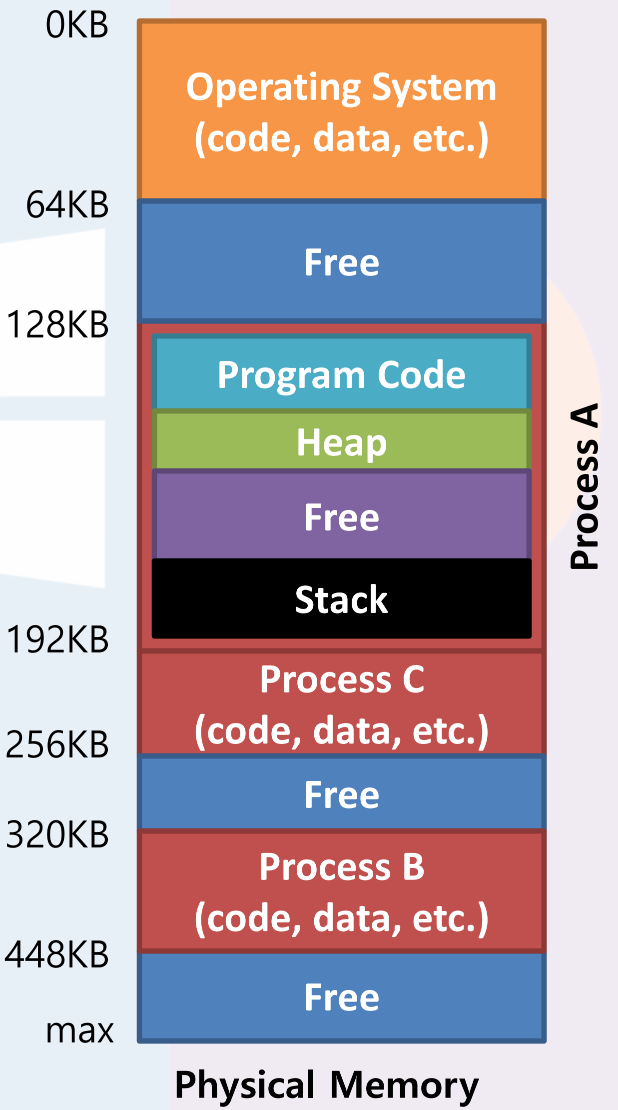
- **The OS must take action to implement base-and-bounds approach**

- **Three critical junctures:**
  - When a **process starts running**:
    - Finding space for address space in physical memory
  - When a **process is terminated**:
    - Reclaiming the memory for use
  - When **context switch occurs**:
    - Saving and storing the base-and-bounds pair

OS Issues: When a Process Starts Running

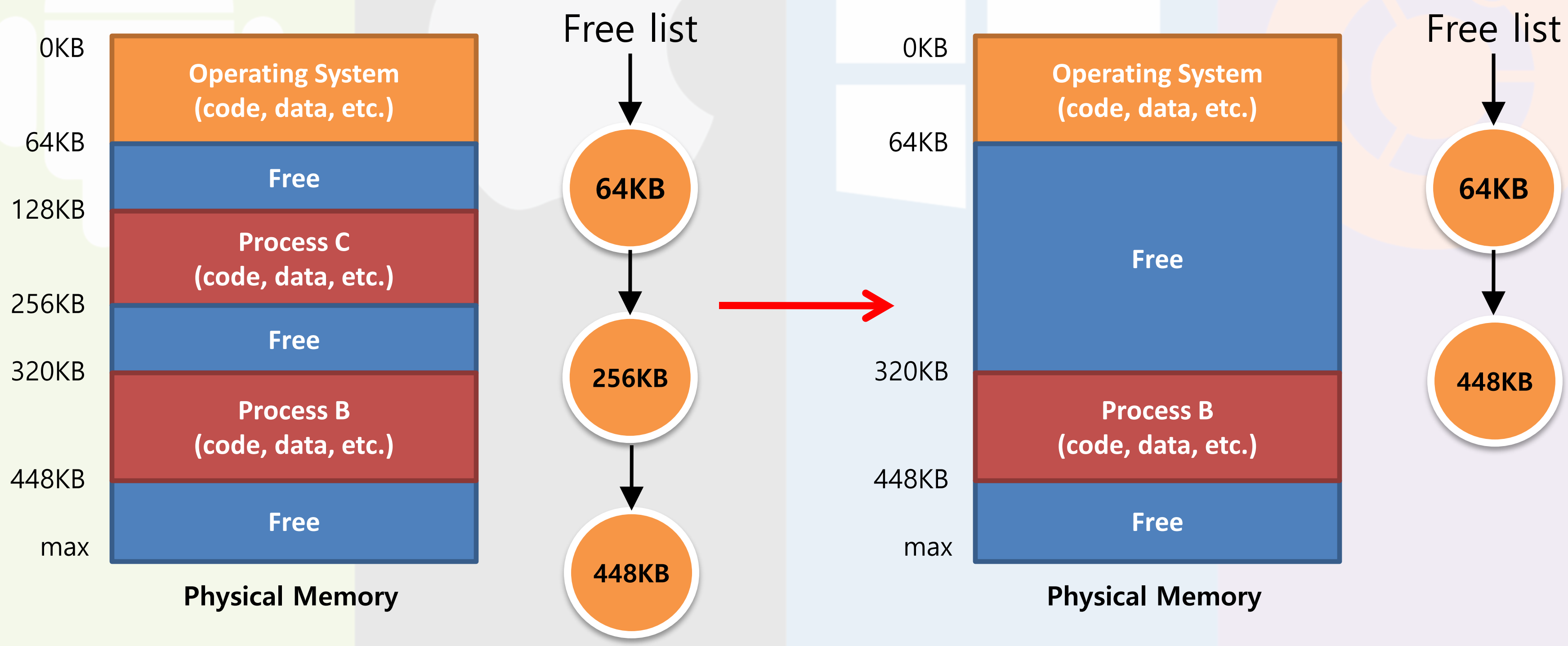
- **The OS must find a room for a new address space**
  - **free list :** A list of the range of the physical memory which are not in use





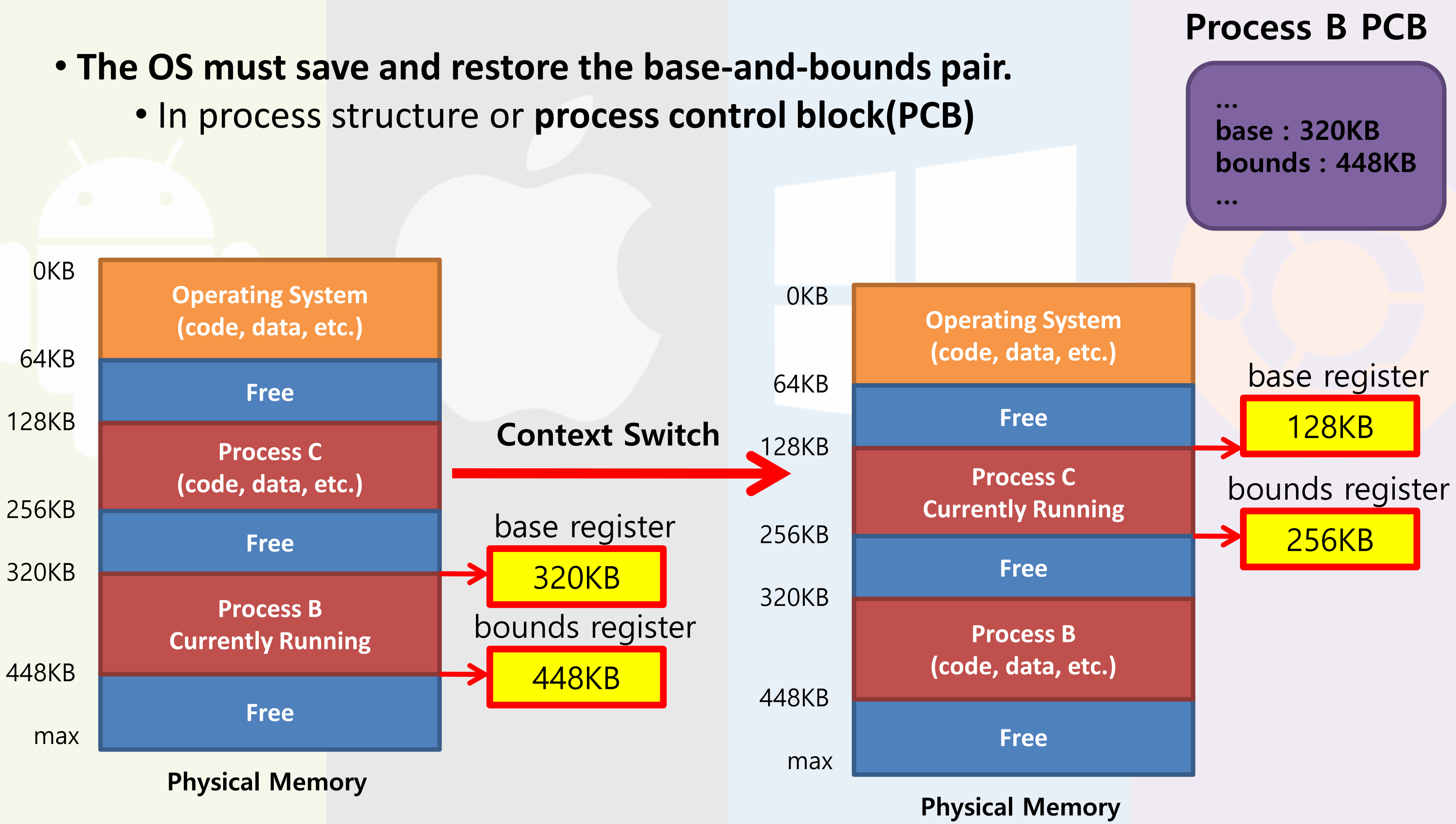
OS Issues: When a Process Is Terminated

- The OS must put the memory back on the free list



OS Issues: When Context Switch Occurs

- **The OS must save and restore the base-and-bounds pair.**
  - In process structure or **process control block(PCB)**

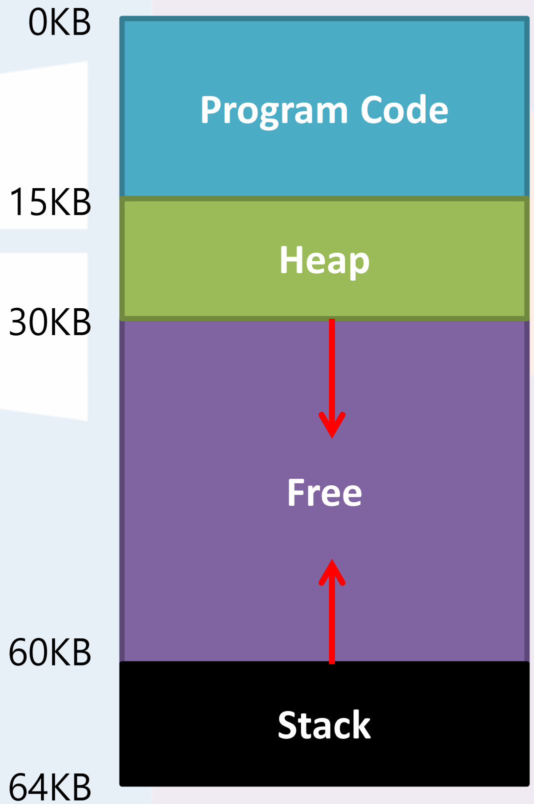


What if there is not enough space together to fit a program?

- **How do we start a new program in this case**

Inefficiency of the Base and Bound Approach

- Big chunk of “free” space
- “free” space takes up physical memory.
- Hard to run when an address space does not fit into physical memory

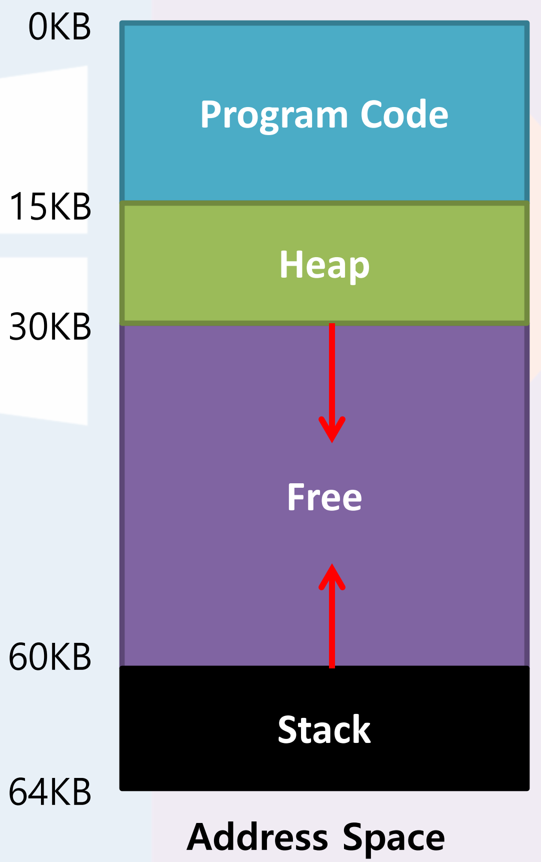


Address Space

Segmentation

- **Segment is just a contiguous portion of the address space of a particular length**
  - Logically-different segment: code, stack, heap

- **Each segment can be placed in different part of physical memory**
  - Base and bounds exist per each segment

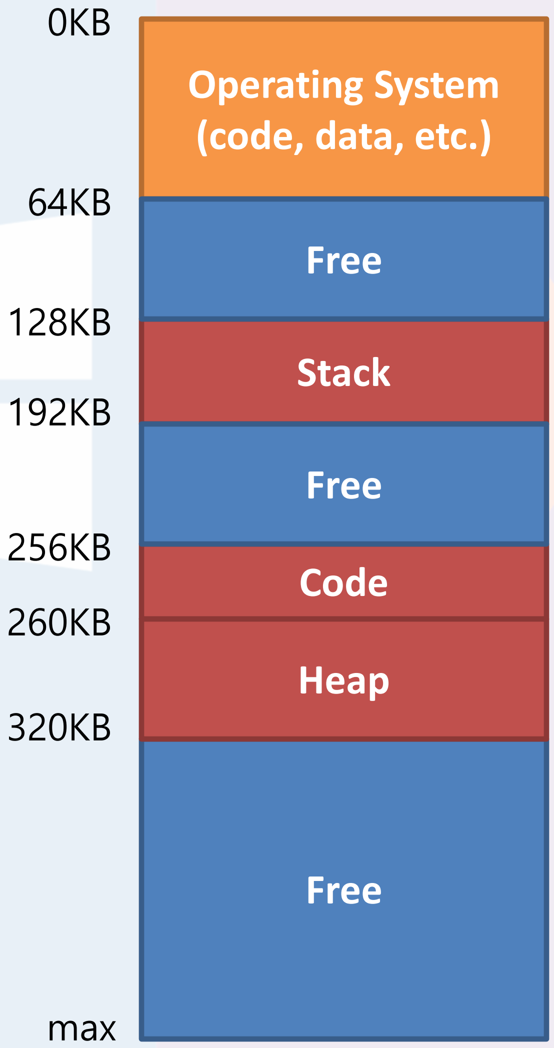


Segmentation

- **Segment is just a contiguous portion of the address space of a particular length**
  - Logically-different segment: code, stack, heap
- **Each segment can be placed in different part of physical memory**
  - Base and bounds exist per each segment

Segment Register

Segment	Base	Size
Code	256KB	4KB
Heap	260KB	60KB
Stack	128KB	64KB



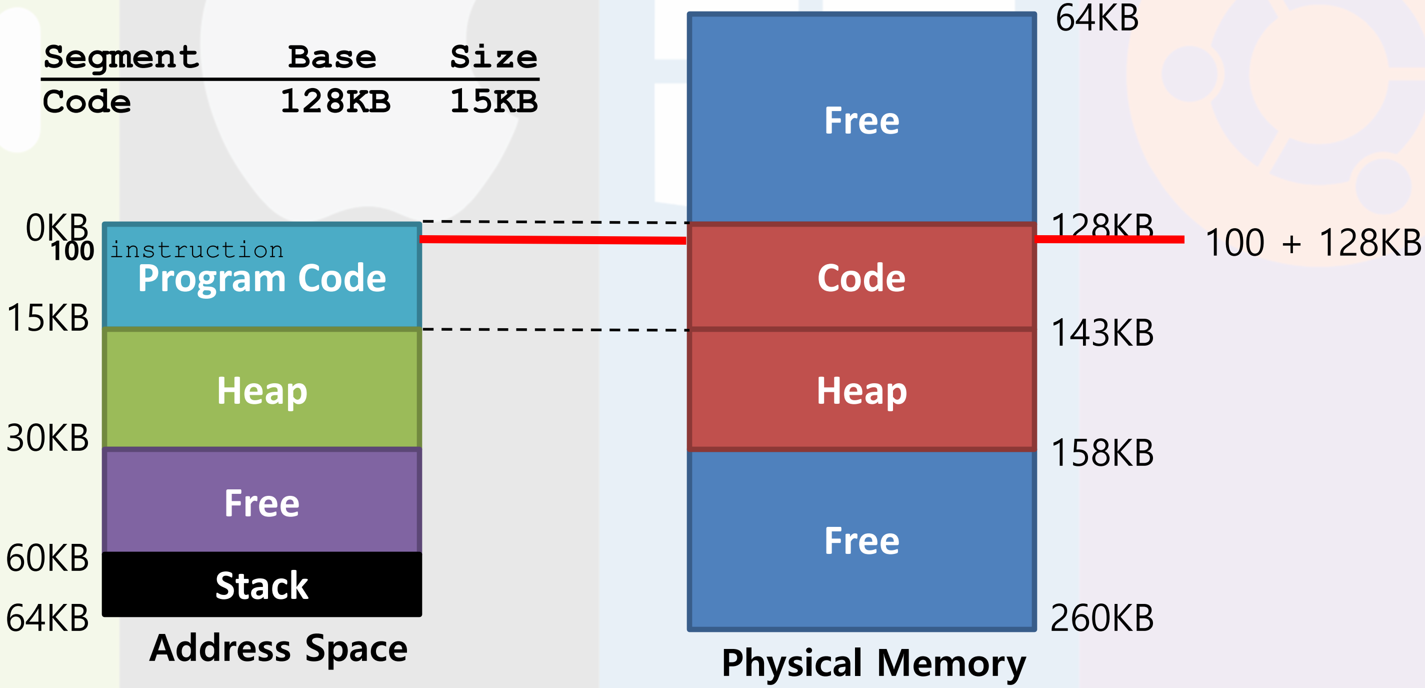
Physical Memory

Address Translation with Segmentation

Physical address = offset + base

- **The offset of virtual address 100 is 100.**
  - The code segment starts at virtual address 0 in address space.

Segment	Base	Size
Code	128KB	15KB

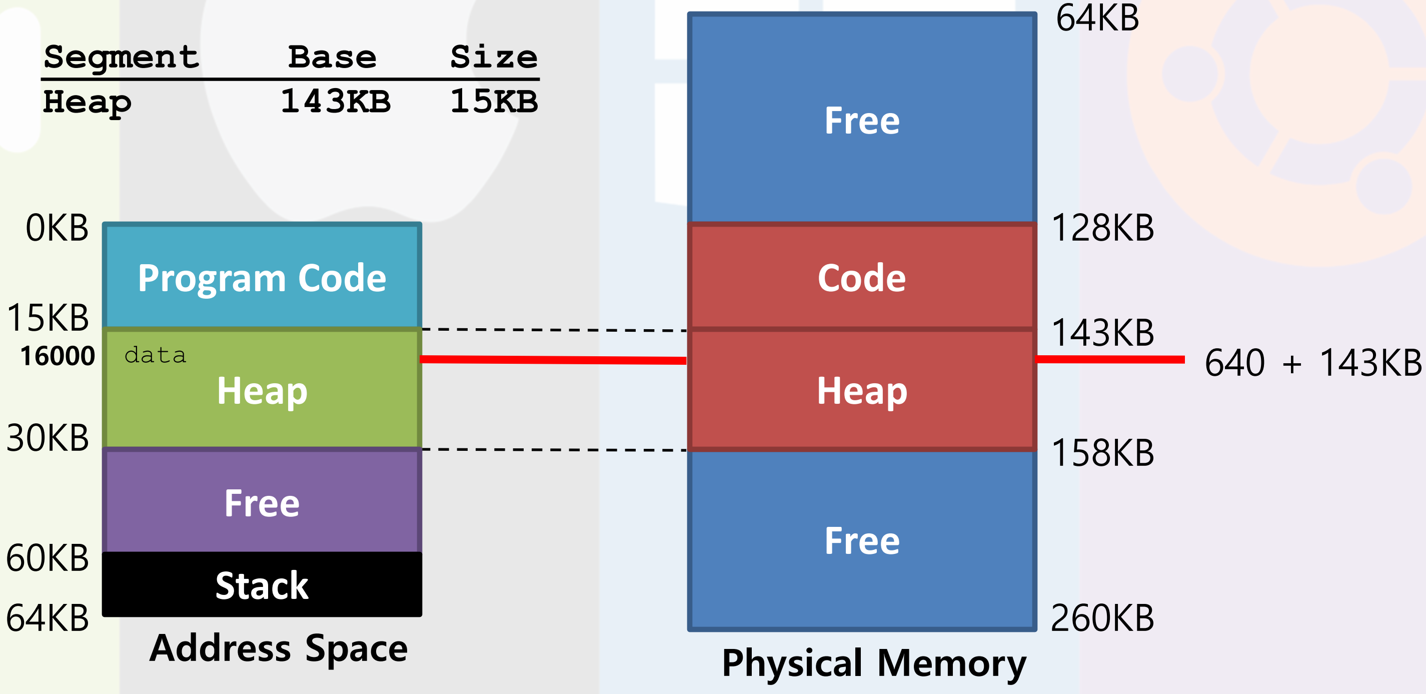


Address Translation with Segmentation

Virtual address + base is not the correct physical address

- **The offset of virtual address 16000 is 640.**
  - The code segment starts at virtual address 0 in address space.

Segment	Base	Size
Heap	143KB	15KB

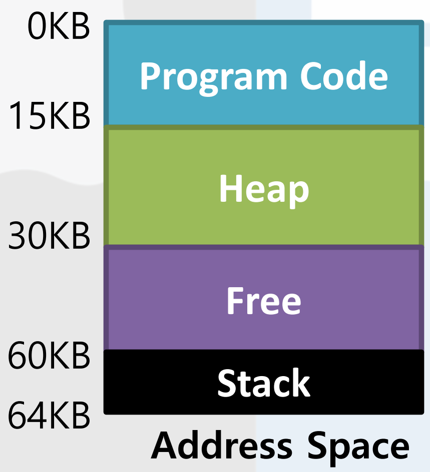


What happens if an address is incorrectly referenced?

- Segmentation Fault occurs

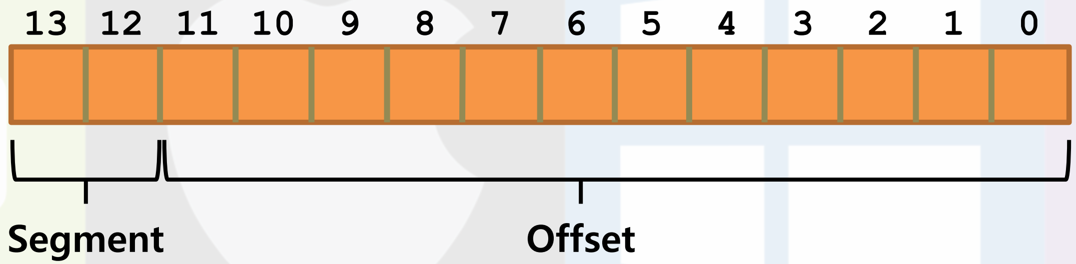
Segmentation Faults

- **If an illegal address such as 37KB which is beyond the end of heap is referenced, the OS occurs segmentation fault**
  - The hardware detects that address is out of bounds



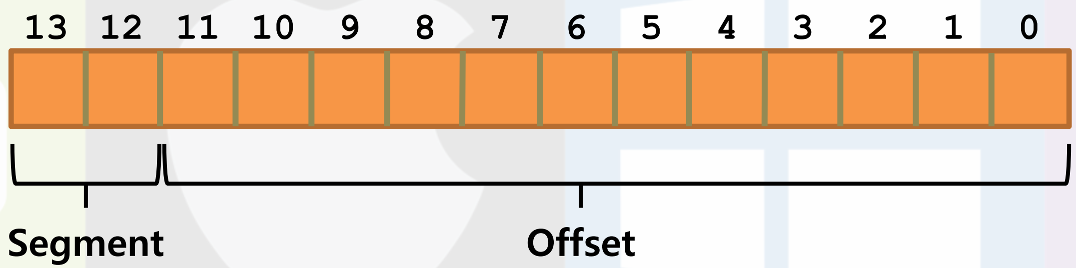
Referring to a Segment

- **Explicit approach**
  - Chop up the address space into segments based on the top few bits of virtual address



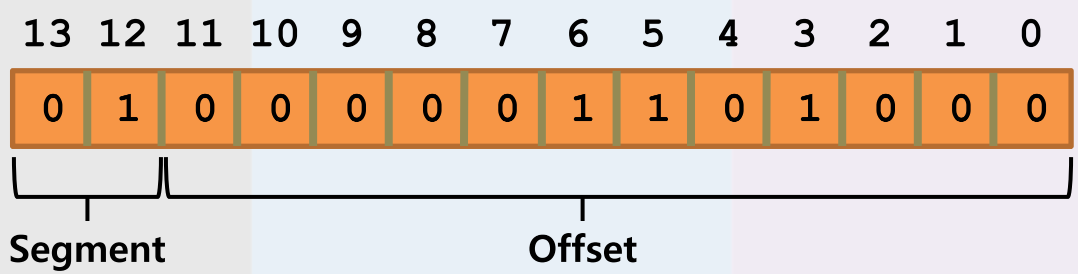
Referring to a Segment

- **Explicit approach**
  - Chop up the address space into segments based on the top few bits of virtual address



- **Example: virtual address 4200 (01000001101000)**

Segment	bits
Code	00
Heap	01
Stack	10
-	11



Referring to a Segment

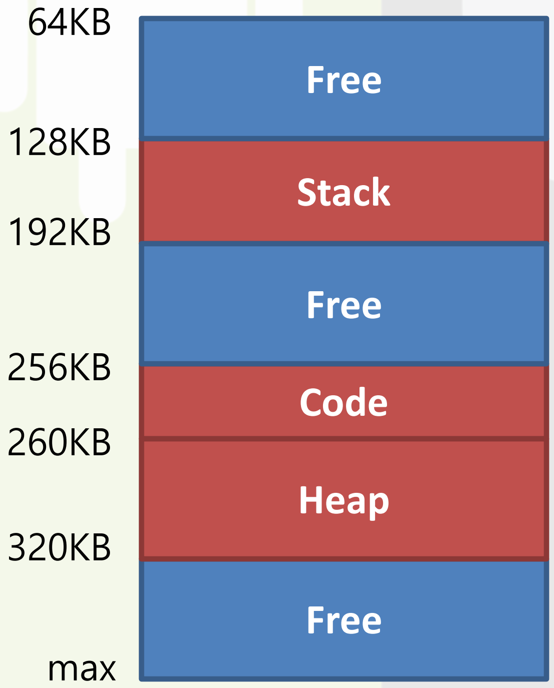
```
// get top 2 bits of 14-bit VA
Segment = (VirtualAddress & SEG_MASK) >> SEG_SHIFT
// now get offset
Offset = VirtualAddress & OFFSET_MASK
if (Offset >= Bounds[Segment])
    RaiseException(PROTECTION_FAULT)
else
    PhysAddr = Base[Segment] + Offset
    Register = AccessMemory(PhysAddr)
```

- **SEG_MASK = 0x3000(11000000000000)**
- **SEG_SHIFT = 12**
- **OFFSET_MASK = 0xFFF (00111111111111)**

- VA = 01100011100011
- Segment = 01000000000000 >> 12 = 01
- Offset = 00100011100011

Referring to a Stack Segment

- **Stack grows backward**
- **Extra hardware support is need**
  - The hardware checks which way the segment grows
  - 1: positive direction
  - 0: negative direction



Physical Memory

Segment Register(with Negative-Growth Support)

Segment	Base	Size	Grows Positive?
Code	256KB	4KB	1
Heap	260KB	60KB	1
Stack	192KB	64KB	0

Support for Sharing

- **Segment can be shared between address space**
  - Code sharing is still in use in systems today
  - by extra hardware support

- **Extra hardware support is need for form of Protection bits**
  - A few more bits per segment to indicate permissions of read, write and execute

Segment Register(with Negative-Growth Support)

Segment	Base	Size	Grows Positive?	Protection
Code	256KB	4KB	1	Read-Execute
Heap	260KB	60KB	1	Read-Write
Stack	192KB	64KB	0	Read-Write

Fine-Grained and Coarse-Grained

- **Coarse-Grained means segmentation in a small number**
  - e.g., code, heap, stack

- **Fine-Grained segmentation allows more flexibility for address space in some early system**
  - To support many segments, Hardware support with a segment table is required

OS support: Fragmentation

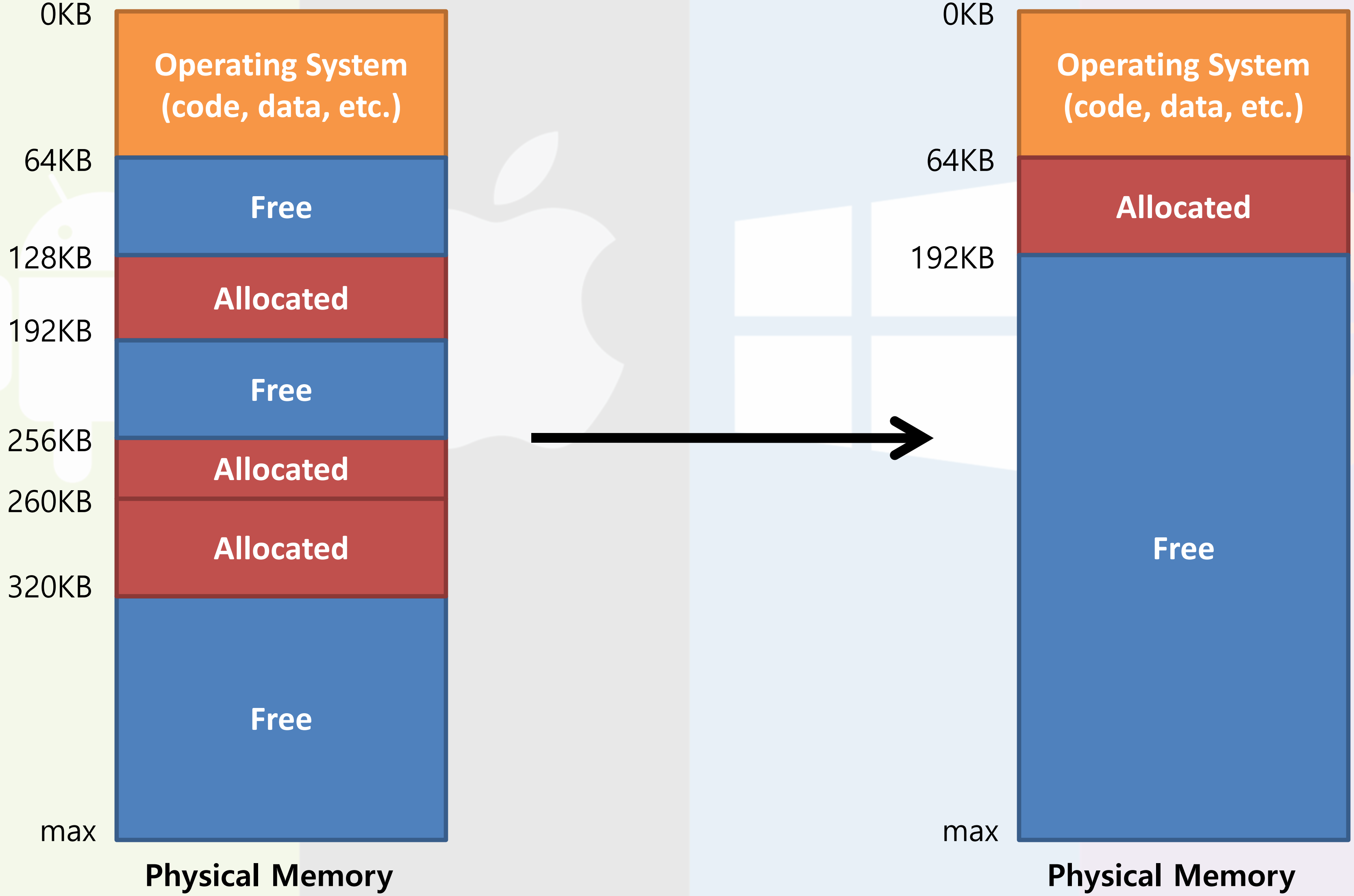
- **External Fragmentation: little holes of free space in physical memory that make difficulty to allocate new segments**
  - There is 24KB free, but not in one contiguous segment
  - The OS cannot satisfy the 20KB request

OS support: Fragmentation

- **External Fragmentation: little holes of free space in physical memory that make difficulty to allocate new segments**
    - There is 24KB free, but not in one contiguous segment
    - The OS cannot satisfy the 20KB request

- **Compaction: rearranging the existing segments in physical memory**
    - Compaction is costly
        - Stop running process
        - Copy data to somewhere
        - Change segment register value

Memory Compaction



Free-Space Management

- Splitting
- Coalescing
- Keeping Track of Memory Region Sizes
- Strategies for Managing Memory

Splitting

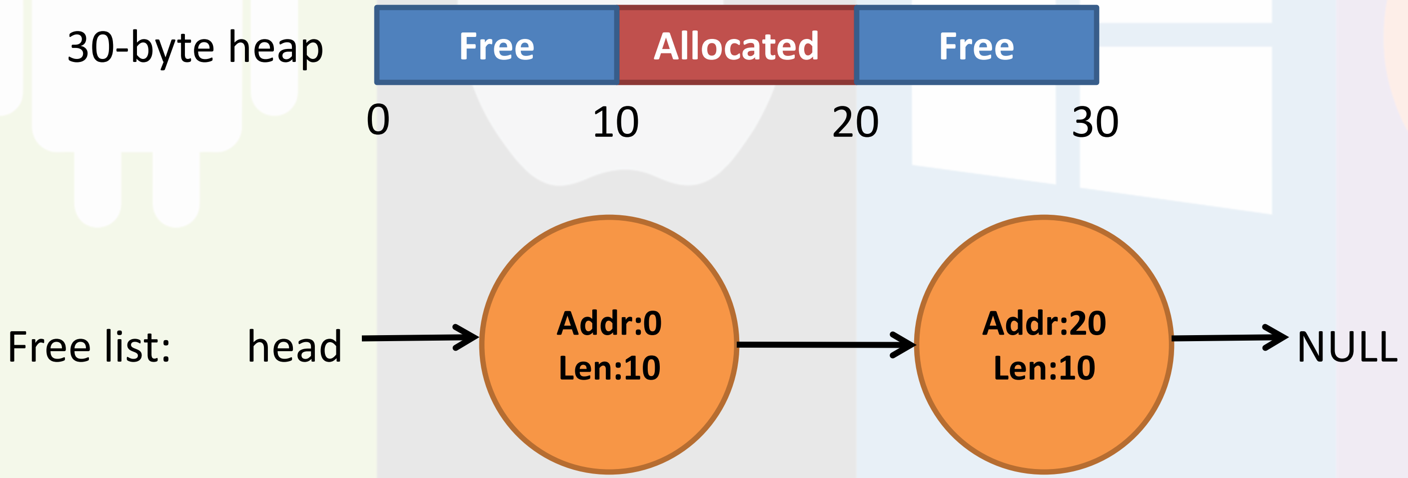
- Finding a free chunk of memory that can satisfy the request and splitting it into two

Splitting

- **Finding a free chunk of memory that can satisfy the request and splitting it into two**
  - **For Example :** When request for memory allocation is smaller than the size of free chunks.

Splitting

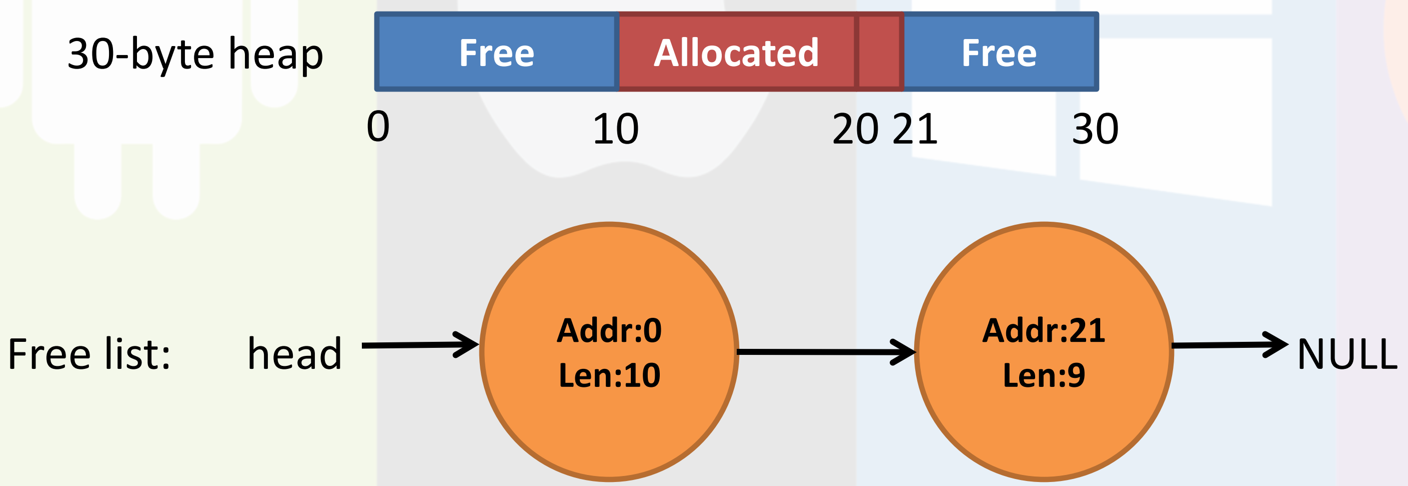
- **For Example :** When request for memory allocation is smaller than the size of free chunks.
  - **Two 10-byte free segments with a 1-byte request**



Before Splitting

Splitting

- **For Example :** When request for memory allocation is smaller than the size of free chunks.
  - **Two 10-byte free segments with a 1-byte request**



After Splitting 10-byte free segment

Coalescing

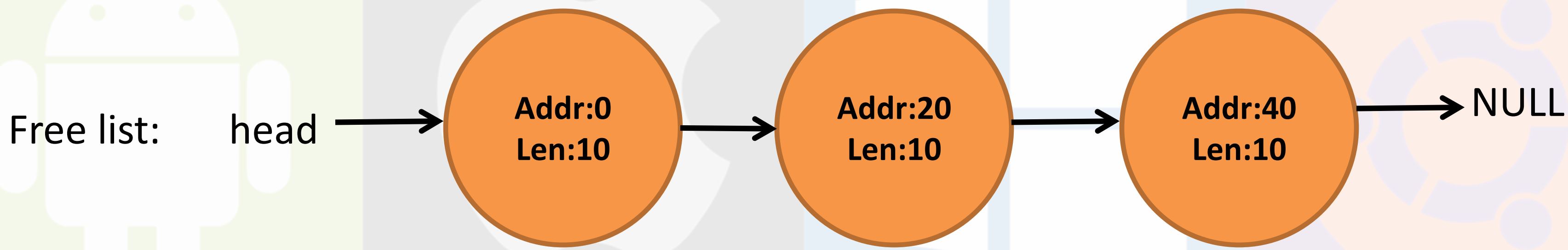
- Merge returning a free chunk with existing chunks into a large single free chunk if addresses of them are nearby

Coalescing

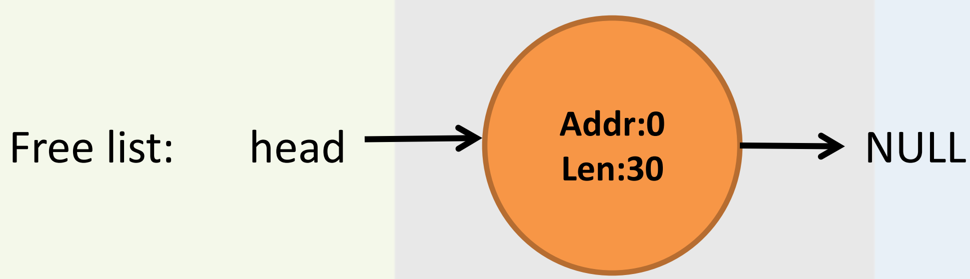
- **Merge returning a free chunk with existing chunks into a large single free chunk if addresses of them are nearby**
  - **For Example :** When request for memory that is bigger than free chunk size, the list will not find such a free chunk

Coalescing

- **For Example :** When request for memory that is bigger than free chunk size, the list will not find such a free chunk
    - **Three 10-byte free segments with a 20-byte request**



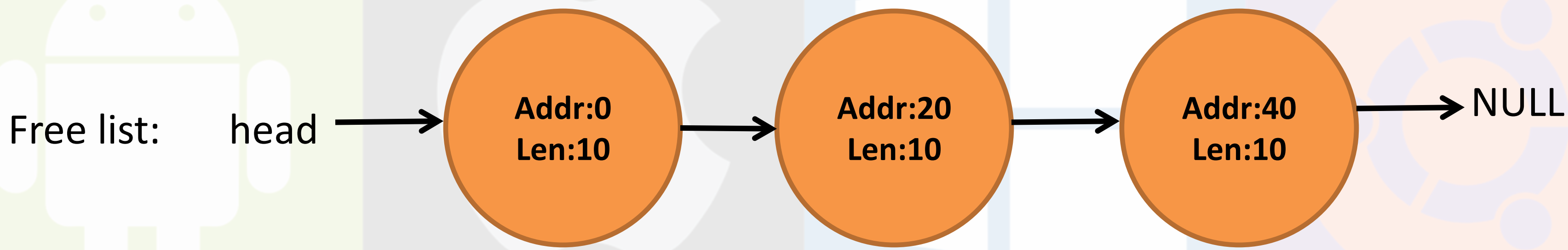
Before Coalescing



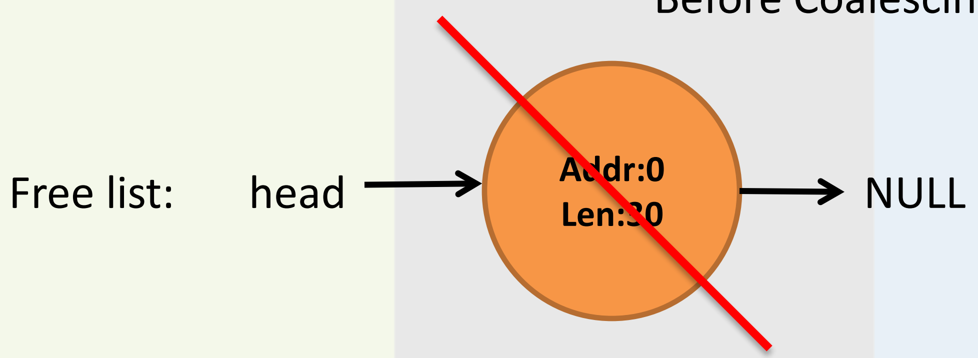
After Coalescing

Coalescing

- **For Example :** When request for memory that is bigger than free chunk size, the list will not find such a free chunk
  - **Three 10-byte free segments with a 20-byte request**



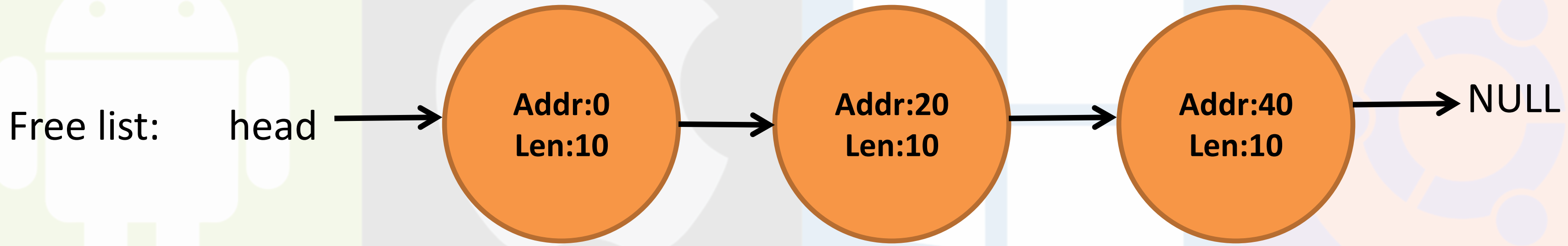
Before Coalescing



After Coalescing

Coalescing

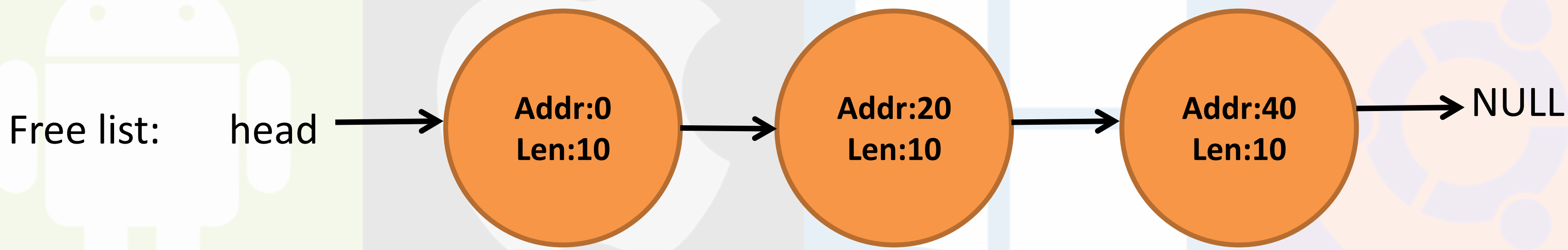
- **For Example :** When request for memory that is bigger than free chunk size, the list will not find such a free chunk
  - **Three 10-byte free segments with a 20-byte request**



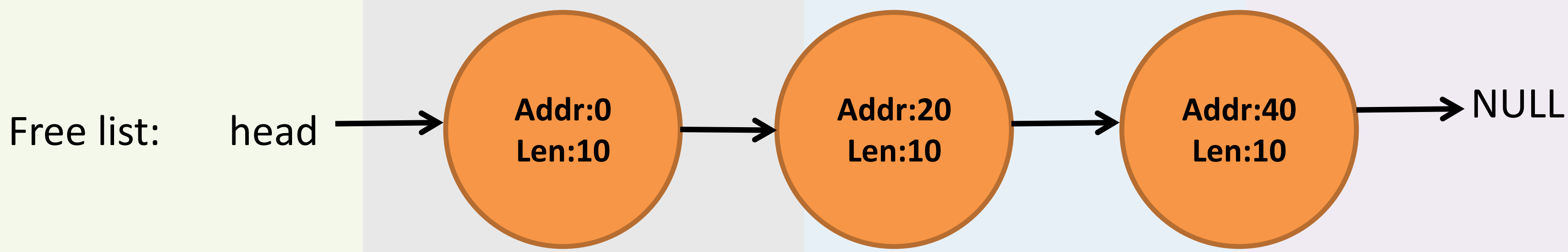
Before Coalescing

Coalescing

- **For Example :** When request for memory that is bigger than free chunk size, the list will not find such a free chunk
  - **Three 10-byte free segments with a 20-byte request**



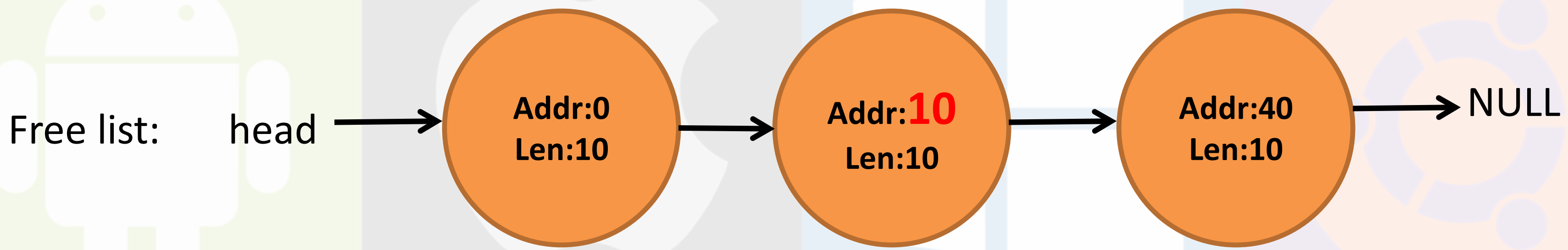
Before Coalescing



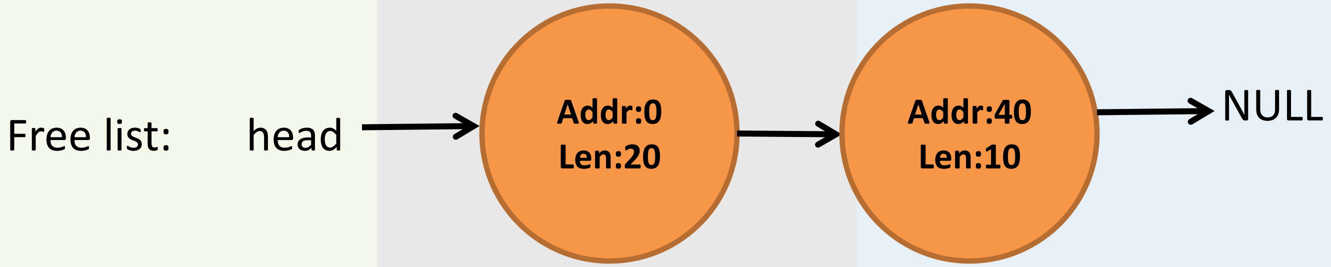
After Coalescing

Coalescing

- **For Example :** When request for memory that is bigger than free chunk size, the list will not find such a free chunk
  - **Three 10-byte free segments with a 20-byte request**



Before Coalescing



After Coalescing

How do we do this reorganisation in an efficient way

- By tracking the size of allocated regions
- By embedding the free list

Tracking the size of allocated regions

- The interface to free(void *ptr) does not take a size parameter.

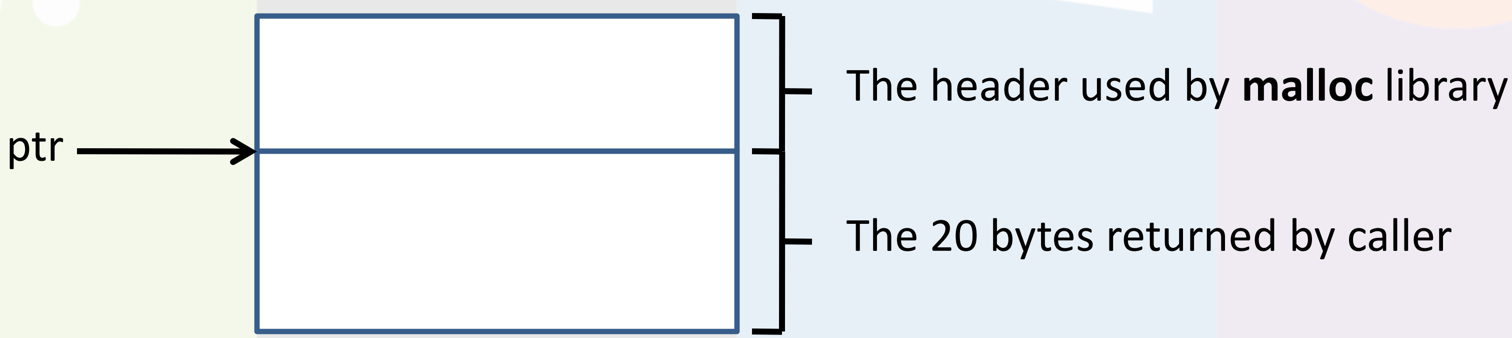
- How does the library know the size of memory region that will be back into free list?

Tracking the size of allocated regions

- **The interface to free(void *ptr) does not take a size parameter.**

- **How does the library know the size of memory region that will be back into free list?**
  - Most allocators store **extra information** in a **header** block

```
ptr = malloc(20);
```

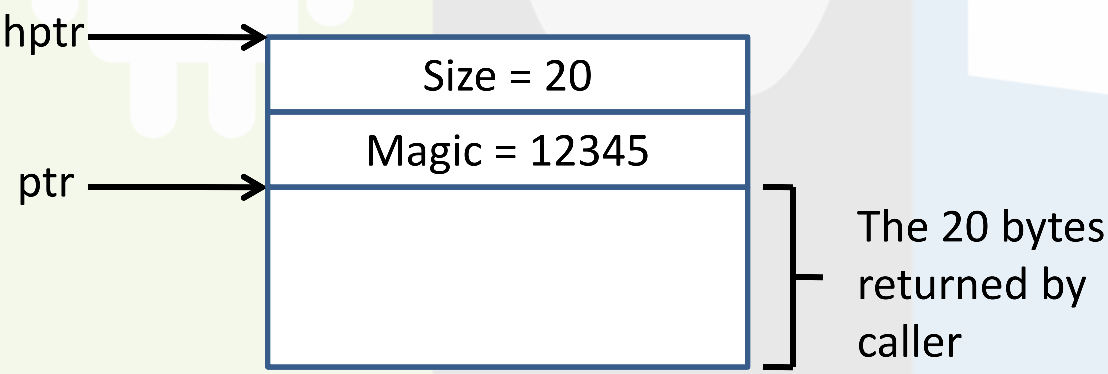


An Allocated Region Plus Header

The header

- **The header minimally contains the size of the allocated memory region**
- **The header may also contain**
  - Additional pointers to speed up deallocation
  - A magic number for integrity checking

ptr = malloc(20);



An Allocated Region Plus Header

```
typedef struct __header_t {
        int size;
        int magic;
} header_t;
```

Simple Header

The header

- **The size for required free region is the size of the header plus the size of the space allocated to the user.**
    - If a user **request N bytes**, the library searches for a free chunk of **size N plus the size of the header**

The header

- **The size for required free region is the size of the header plus the size of the space allocated to the user.**
  - If a user **request N bytes**, the library searches for a free chunk of **size N plus the size of the header**

- **Simple pointer arithmetic to find the header pointer**

```
void free(void *ptr) {
        header_t *hptr = (void *)ptr – sizeof(header_t);
}
```

Embedding a Free List

- The memory-allocation library initializes the heap and puts the first element of the free list in the free space

- But we have a little problem here

Embedding a Free List

- **The memory-allocation library initializes the heap and puts the first element of the free list in the free space**

- **But we have a little problem here**
  - The library that is required by **malloc** to actually allocate memory cant use **malloc** itself.

Embedding a Free List

- **The memory-allocation library initializes the heap and puts the first element of the free list in the free space**

- **But we have a little problem here**
  - The library that is required by **malloc** to actually allocate memory cant use **malloc** itself.

- **Therefore we embed the free list directly in the heap**

Embedding a Free List

- **The memory-allocation library initializes the heap and puts the first element of the free list in the free space**

- **But we have a little problem here**
  - The library that is required by **malloc** to actually allocate memory cant use **malloc** itself.

- **Therefore we embed the free list directly in the heap**

```
typedef struct __node_t {
        int size;
        struct __node_t *next;
} nodet_t;
```

Description of a node of the free list

Embedding a Free List

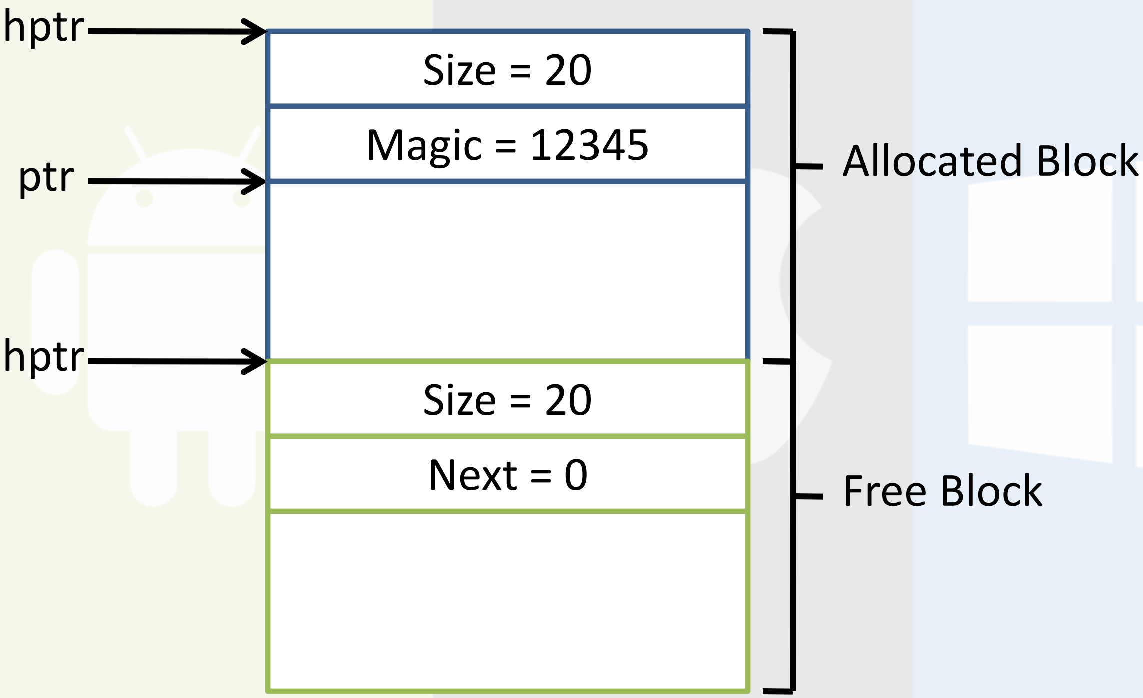
```
typedef struct __header_t {
        int size;
        int magic;
} header_t;
```

Simple Header for an allocated block

```
typedef struct __node_t {
        int size;
        struct __node_t *next;
} nodet_t;
```

Description of a node of the free list

Embedding a Free List



Embedding a Free List : Allocation

- If a chunk of memory is requested, the library will first find a chunk that is large enough to accommodate the request

- This is done by iterating through the free list and checking the size

Embedding a Free List : Allocation

- **If a chunk of memory is requested, the library will first find a chunk that is large enough to accommodate the request**

- **This is done by iterating through the free list and checking the size**

- The library will:
    - **Split** the large free chunk into two.
        - **One** for the **request** and the other being the **remaining** free chunk
    - **Shrink** the size of free chunk in the list

Embedding a Free List : Allocation

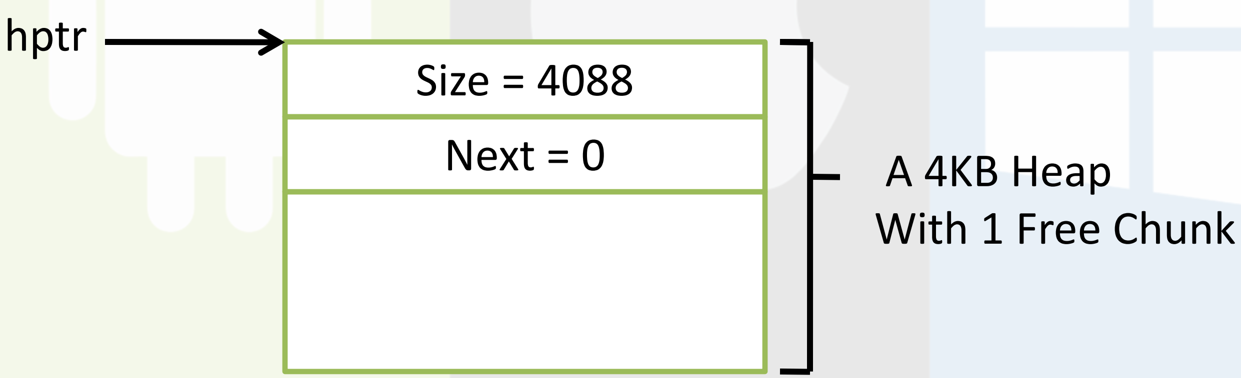
- **Example** a request for 100 bytes by ptr = malloc(100)

Embedding a Free List : Allocation

- **Example** a request for 100 bytes by ptr = malloc(100)
  - Allocating 108 bytes out of the existing one free chunk.
  - shrinking the one free chunk to 3980(4088 minus 108)

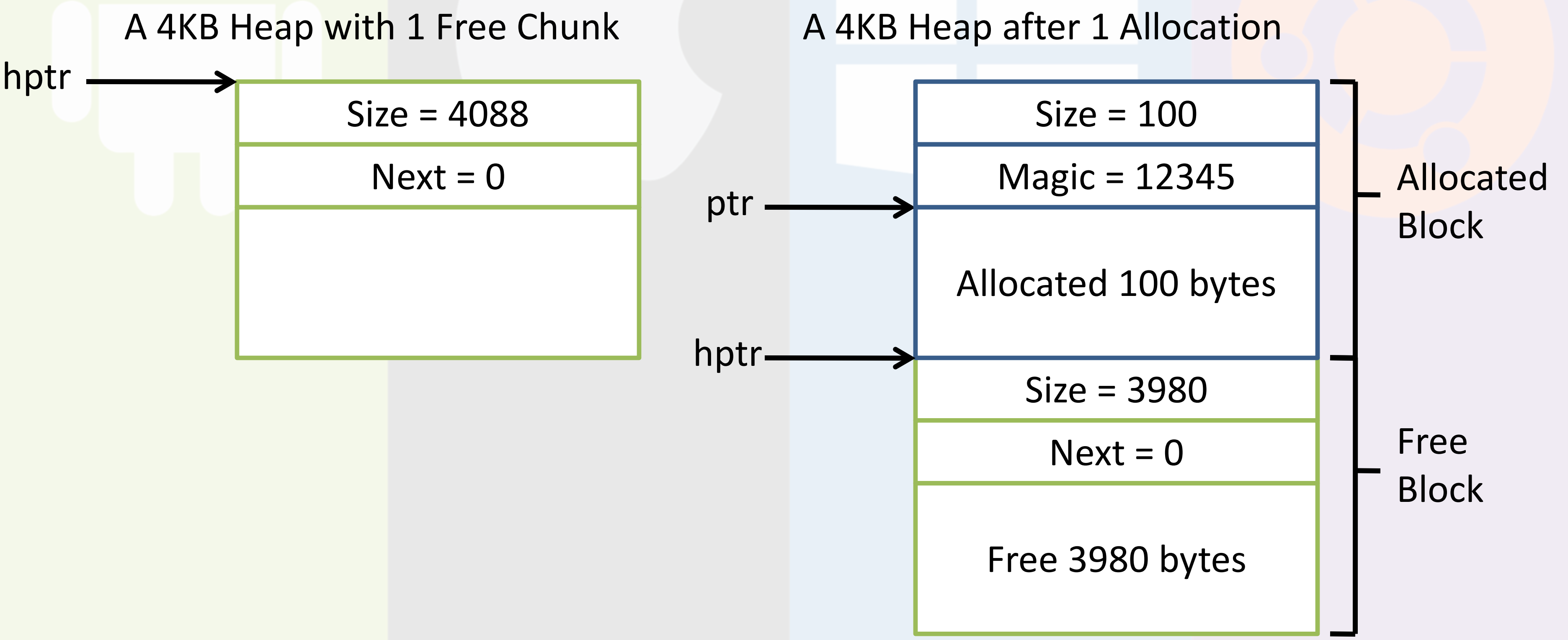
Embedding a Free List : Allocation

- **Example** a request for 100 bytes by ptr = malloc(100)
  - Allocating 108 bytes out of the existing one free chunk.
  - shrinking the one free chunk to 3980(4088 minus 108)

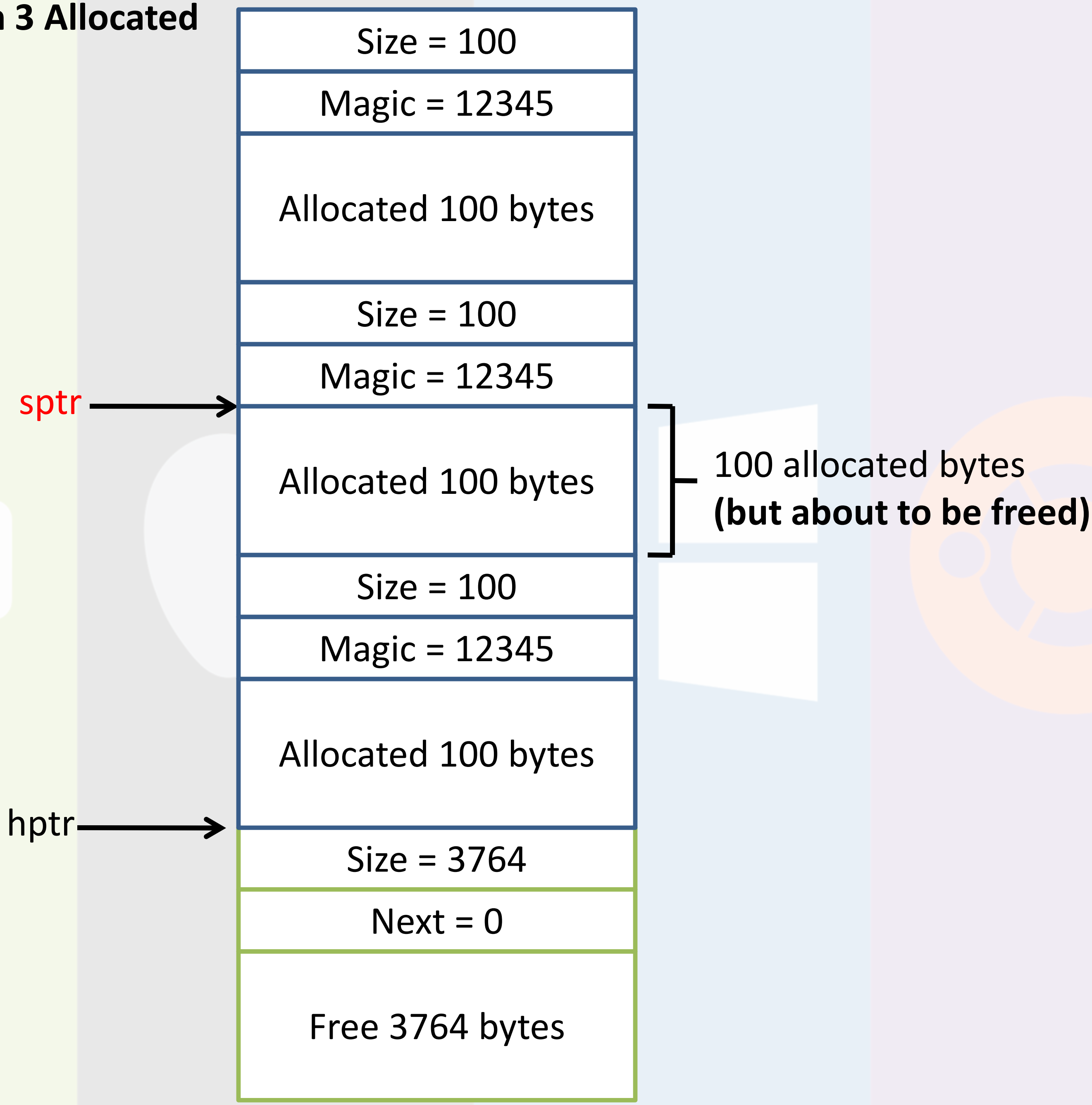


Embedding a Free List : Allocation

- **Example** a request for 100 bytes by ptr = malloc(100)
  - Allocating 108 bytes out of the existing one free chunk.
  - shrinking the one free chunk to 3980(4088 minus 108)



A 4KB Heap with 3 Allocated Chunks



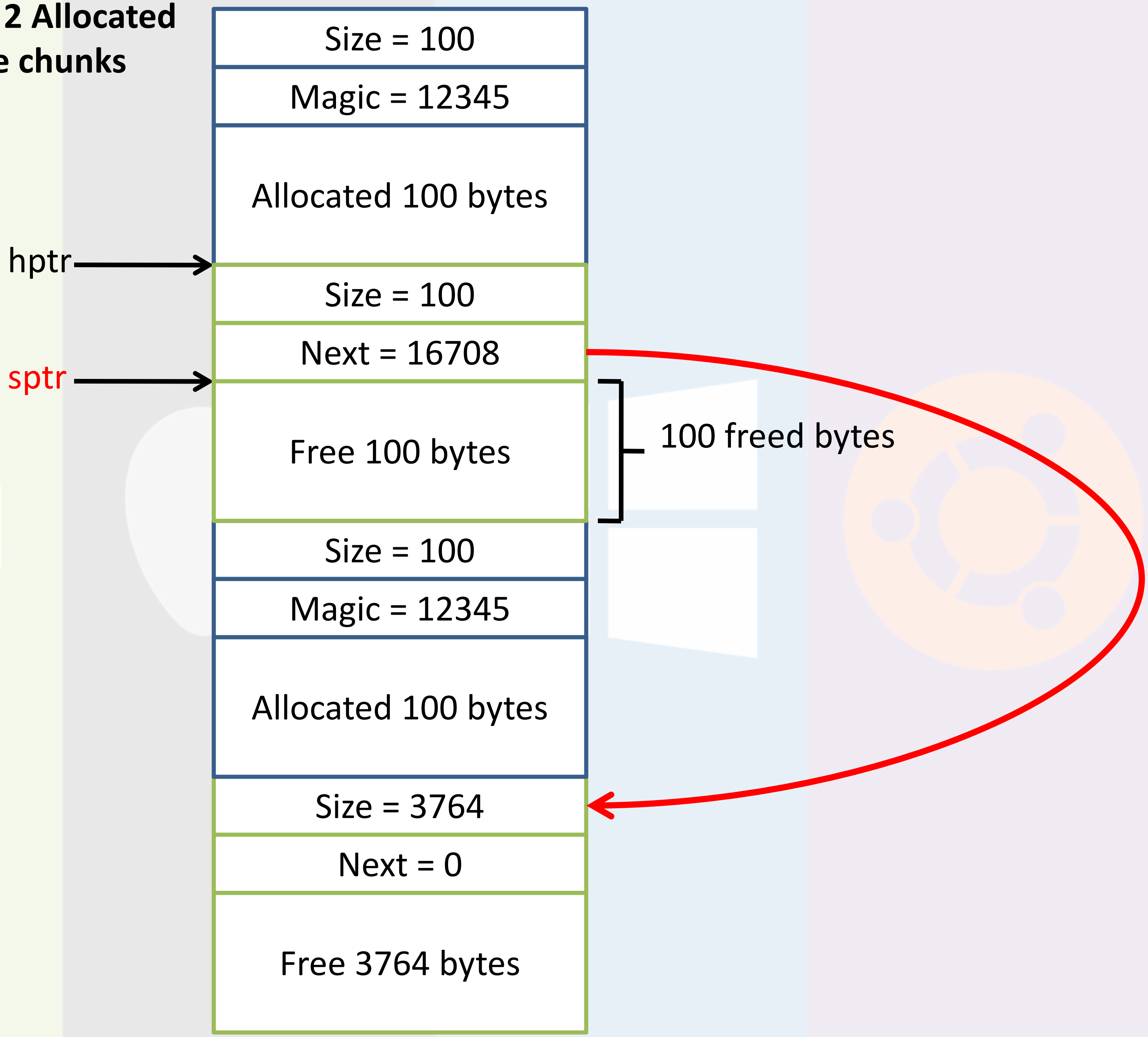
Embedding a Free List : Free

- **Example** a request to free sptr : free(sptr)

Embedding a Free List : Free

- **Example** a request to free sptr : free(sptr)
  - The 100 byte chunk is **back into** the free list.
  - The new Free chunk is added to the front of the list

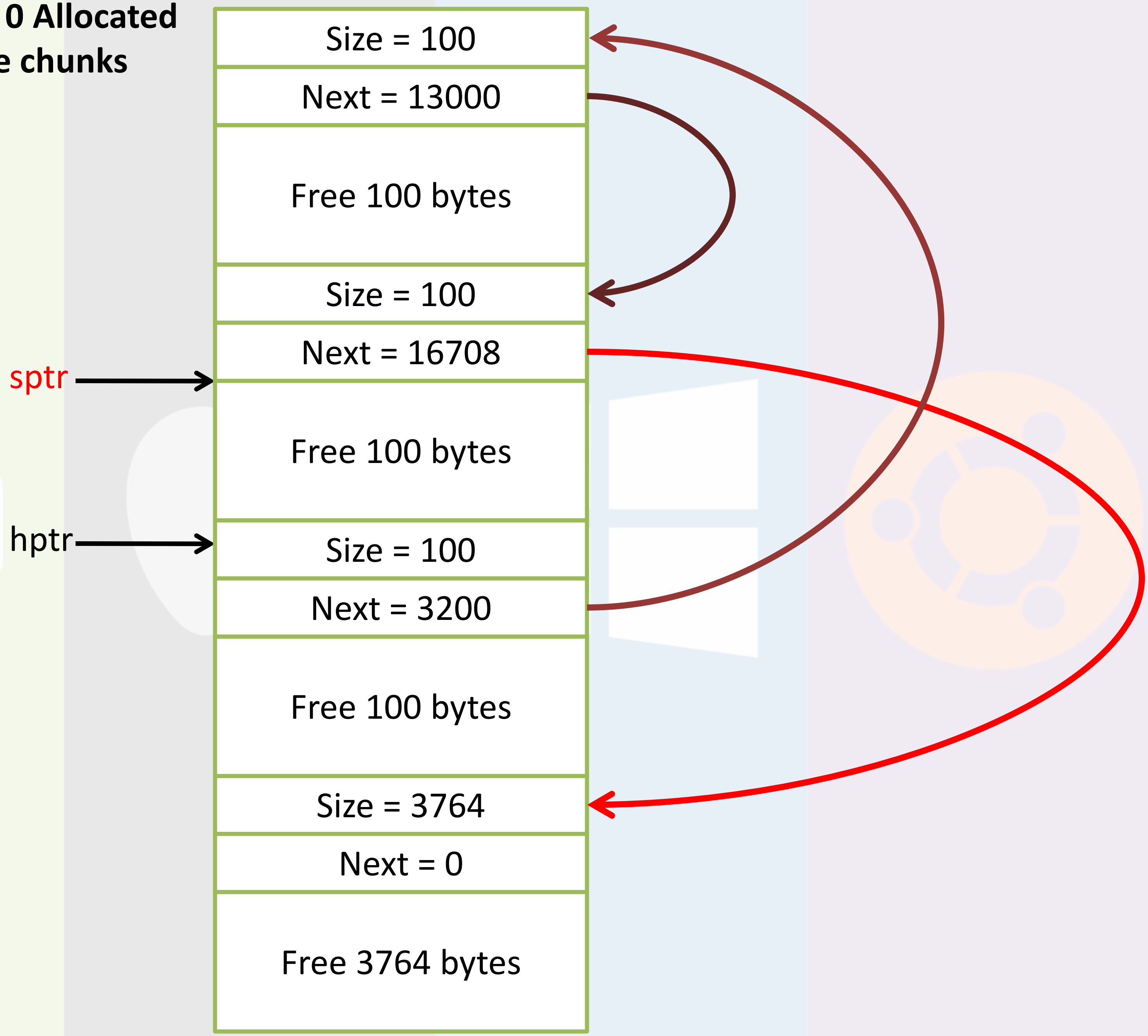
A 4KB Heap with 2 Allocated Chunks and 2 Free chunks



Embedding a Free List : Free

- **Example** Remaining chunks are freed

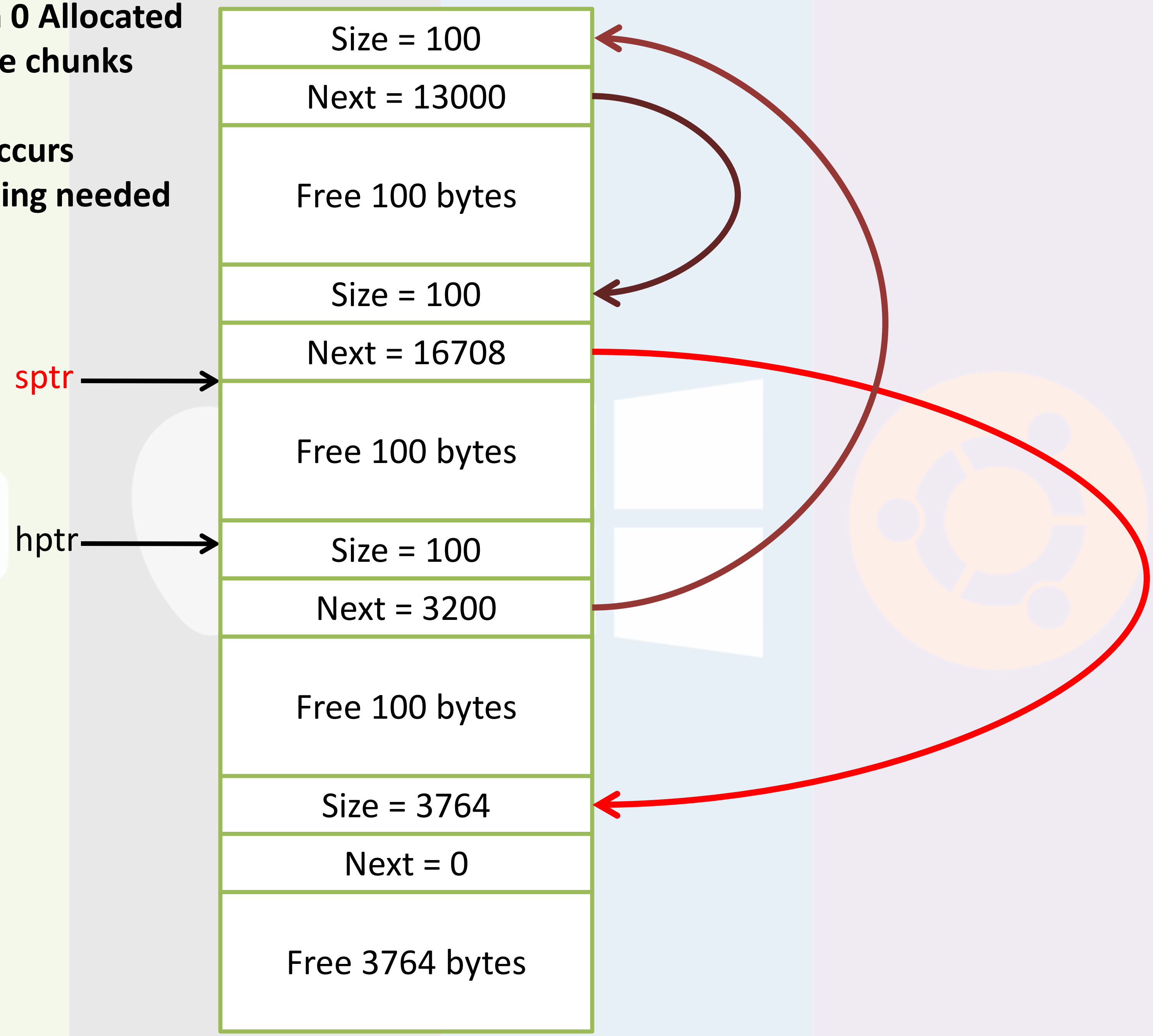
A 4KB Heap with 0 Allocated Chunks and 4 Free chunks



A 4KB Heap with 0 Allocated Chunks and 4 Free chunks

Fragmentation Occurs

Coalescing needed



Managing Free Space : Basic Strategies

- **Best Fit:**
  - Finding free chunks that are **big or bigger than the request**
  - Returning the **one of smallest** chunks **in the group** of candidates

Managing Free Space : Basic Strategies

- **Best Fit:**
  - Finding free chunks that are **equal or bigger than the request**
  - Returning the **one of smallest** in the chunks **in the group** of candidates

- **Worst Fit:**
  - Finding the **largest free chunks** and allocating the amount of the request
  - **Keeping the remaining chunk** on the free list

Managing Free Space : Basic Strategies

- **First Fit:**
  - Finding the **first chunk** that is **big enough** for the request
  - Returning the requested amount and the remaining rest of the chunk

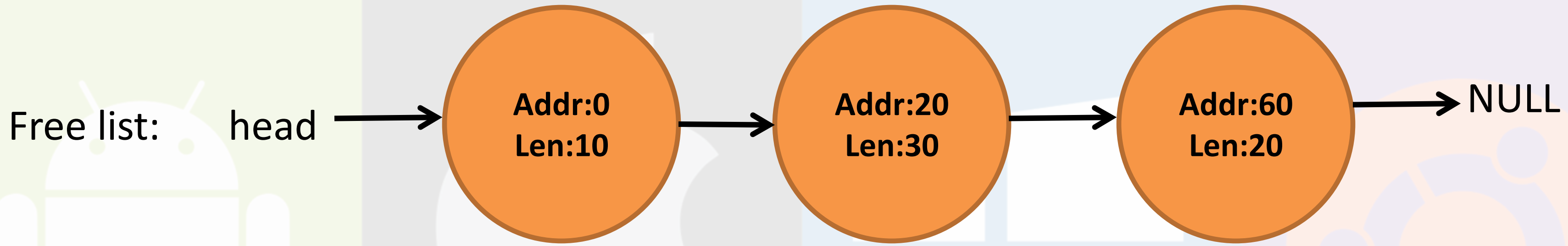
Managing Free Space : Basic Strategies

- **First Fit:**
  - Finding the **first chunk** that is **big enough** for the request
  - Returning the requested amount and the remaining rest of the chunk

- **Next Fit:**
  - Finding the first chunk that is big enough for the request.
  - Searching at **where one was looking** at instead of the begging of the list

Basic Strategies - Examples

- **Allocation Request Size 15**



Basic Strategies - Examples

- **Allocation Request Size 15**

Free list: head → **Addr:0 Len:10** → **Addr:20 Len:30** → **Addr:60 Len:20** → NULL

- **Result of Best-fit**

Free list: head → **Addr:0 Len:10** → **Addr:20 Len:30** → **Addr:75 Len:5** → NULL

Basic Strategies - Examples

- **Allocation Request Size 15**

Free list: head → [Addr:0 Len:10] → [Addr:20 Len:30] → [Addr:60 Len:20] → NULL

- **Result of Best-fit**

Free list: head → [Addr:0 Len:10] → [Addr:20 Len:30] → [Addr:75 Len:5] → NULL

- **Result of Worst-fit**

Free list: head → [Addr:0 Len:10] → [Addr:35 Len:15] → [Addr:60 Len:20] → NULL

Managing Free Space : Buddy Allocation

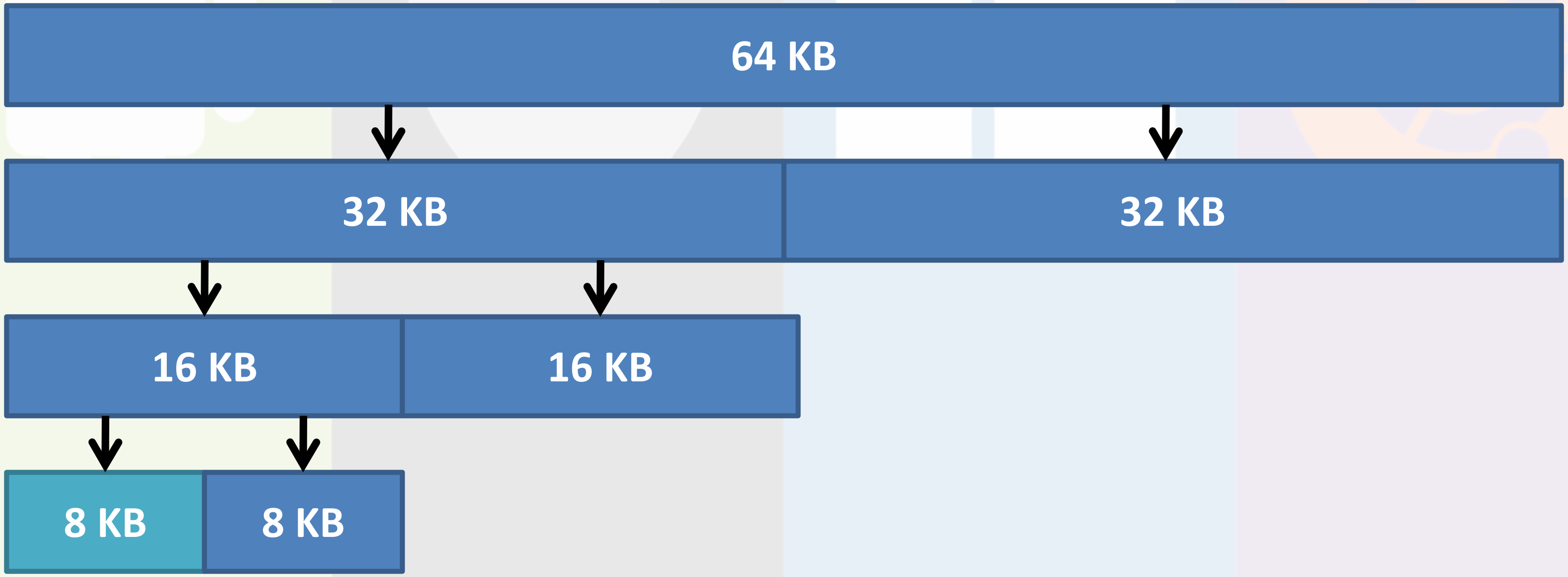
- **Binary Buddy Allocation**

Managing Free Space : Buddy Allocation

- **Binary Buddy Allocation**
  - The allocator **divides free space** by two **until a block** that is big enough to accommodate the request is **found**

Managing Free Space : Buddy Allocation

- **Binary Buddy Allocation**
  - The allocator **divides free space** by two **until a block** that is big enough to accommodate the request is **found**



64KB free space for 7KB request