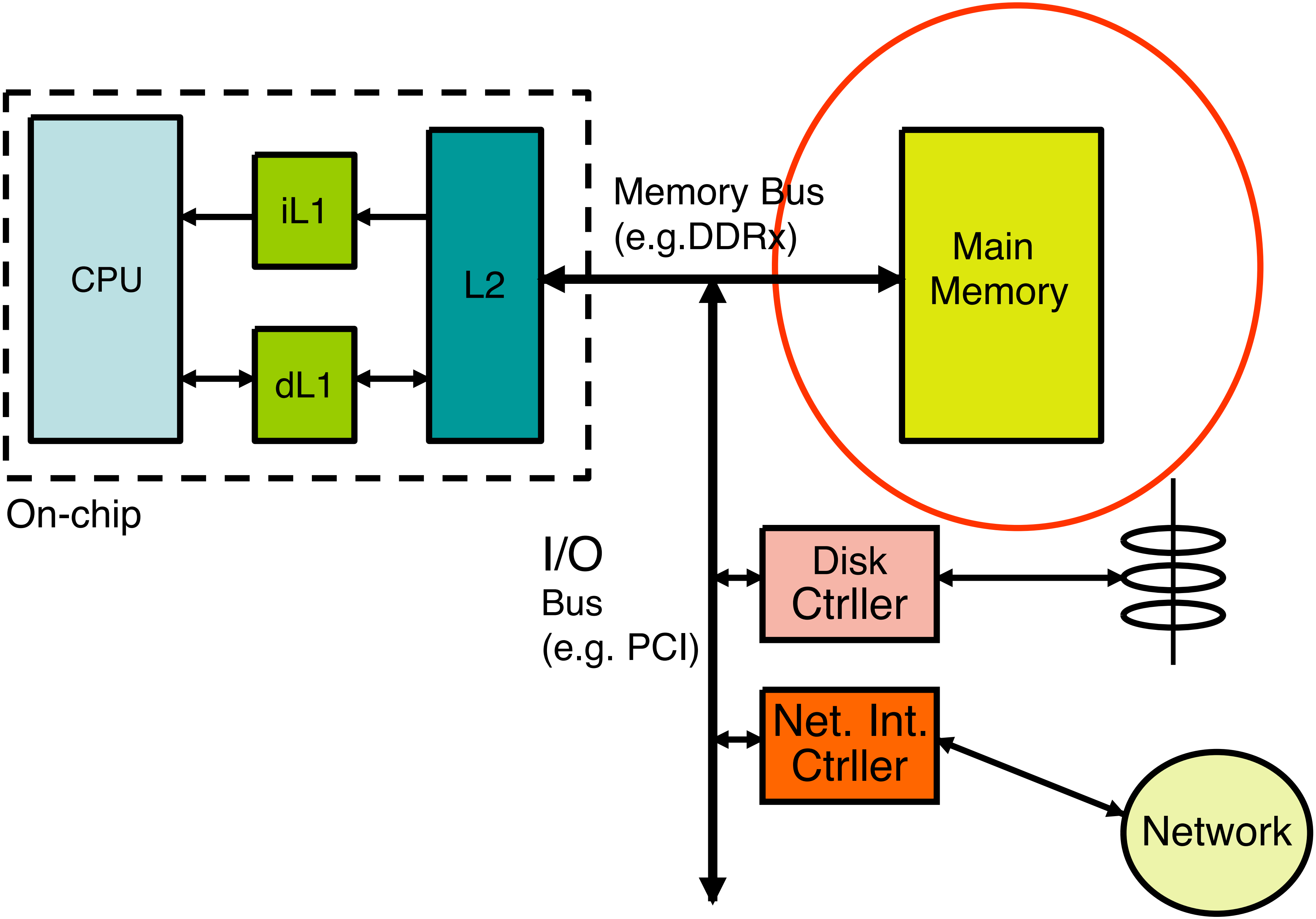
Memory Management

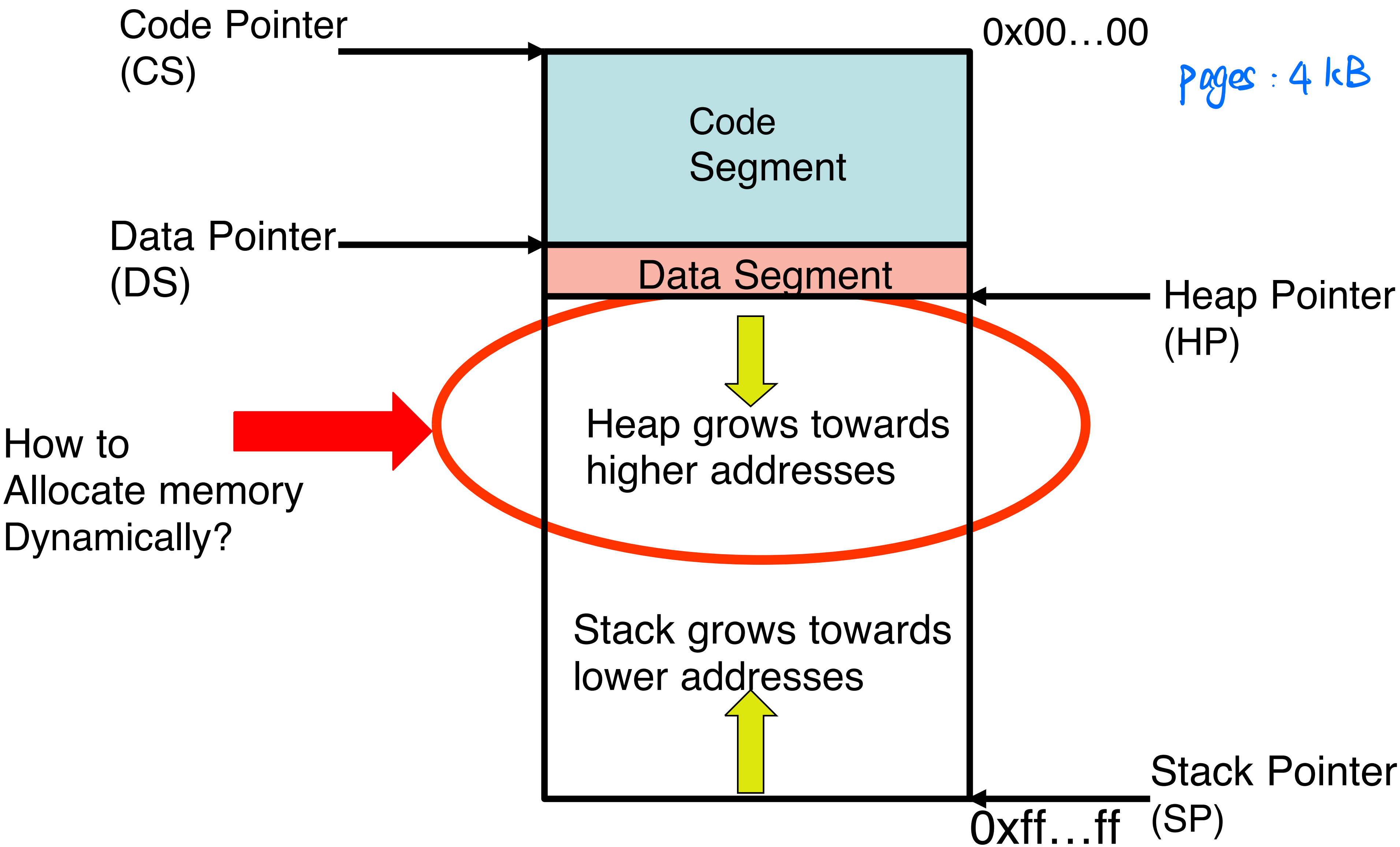

CPU
iL1
dL1
L2
On-chip
Memory Bus
(e.g.DDRx)
Main
Memory
I/O
Bus
(e.g. PCI)
Disk
Ctrller
Net. Int.
Ctrller
Network

Need for Memory Management

- Spatial issues in Allocation
  – How to implement alloc/free?
- Physical memory (DRAM) is limited
  – A single process may not all fit in memory
  – Several processes (their address spaces) also need to fit at the same time
- Disallow 1 process from accessing another processes memory.

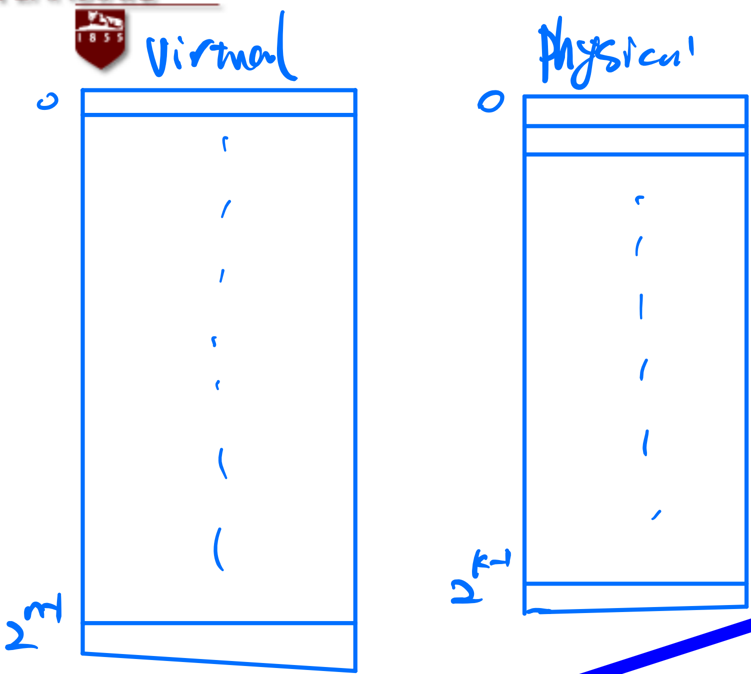
Address Space of a Process

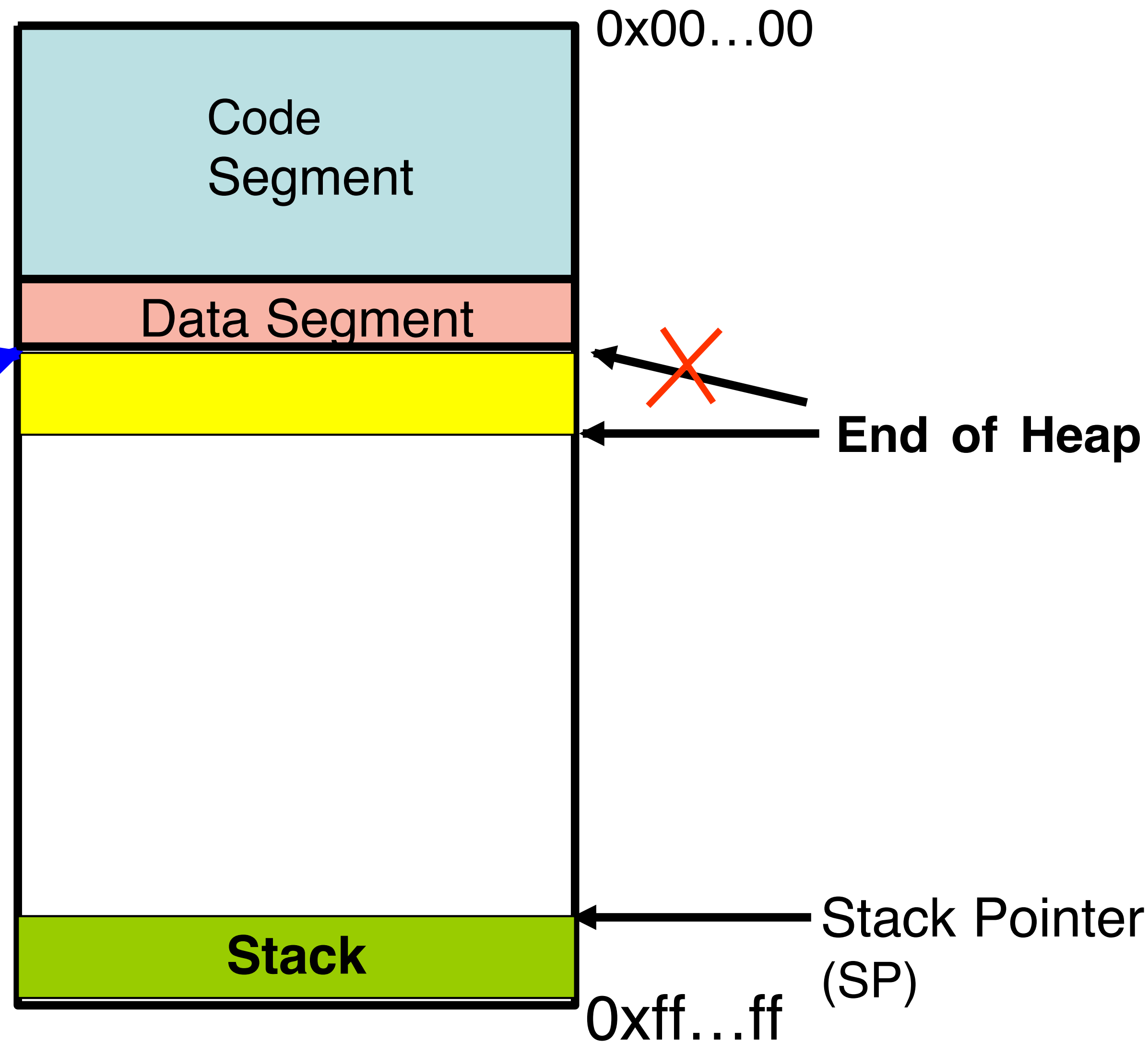
Pages : 4 kB



Heap and Stack are empty before execution!

Malloc()





P1 = malloc(100);

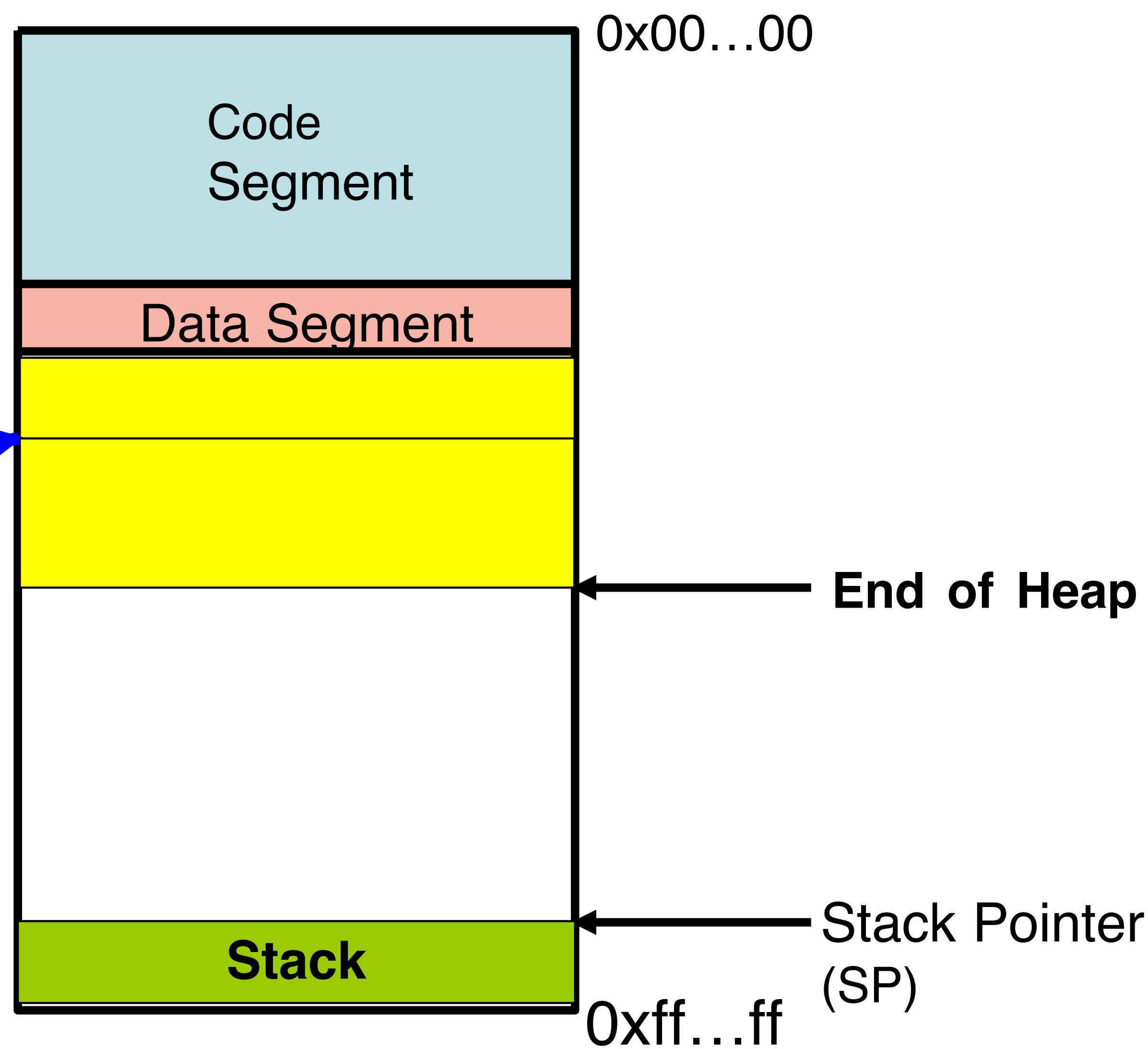
↓ **Library call**

No space in Heap

↓ **Syscall**

sbrk()
(moves end of Heap ptr)

Malloc()



P2 = malloc(200);

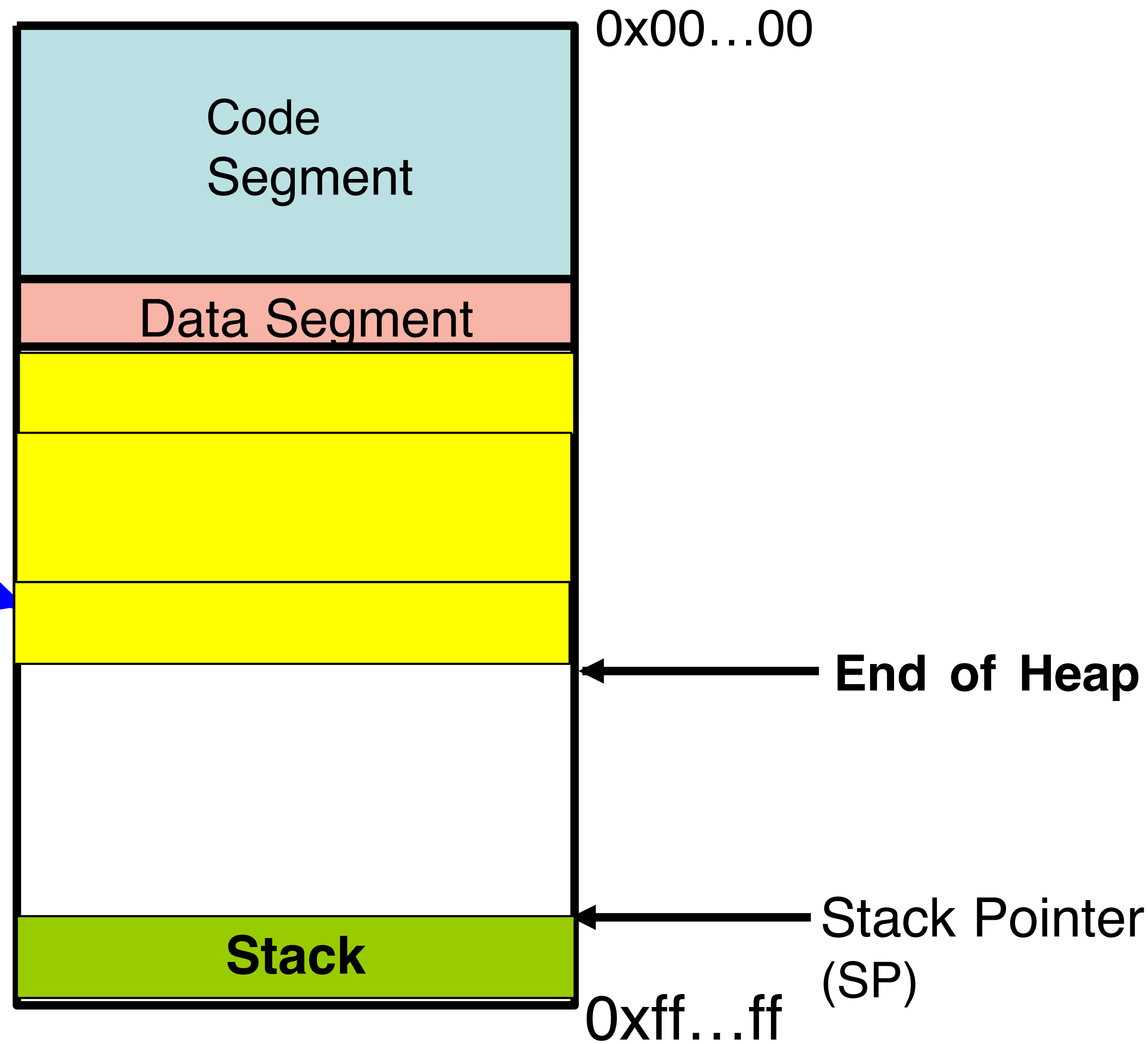
Library call

No space in Heap

Syscall

**sbrk()
(moves end
of Heap ptr)**

Malloc()



P3 = malloc(100);

Library call

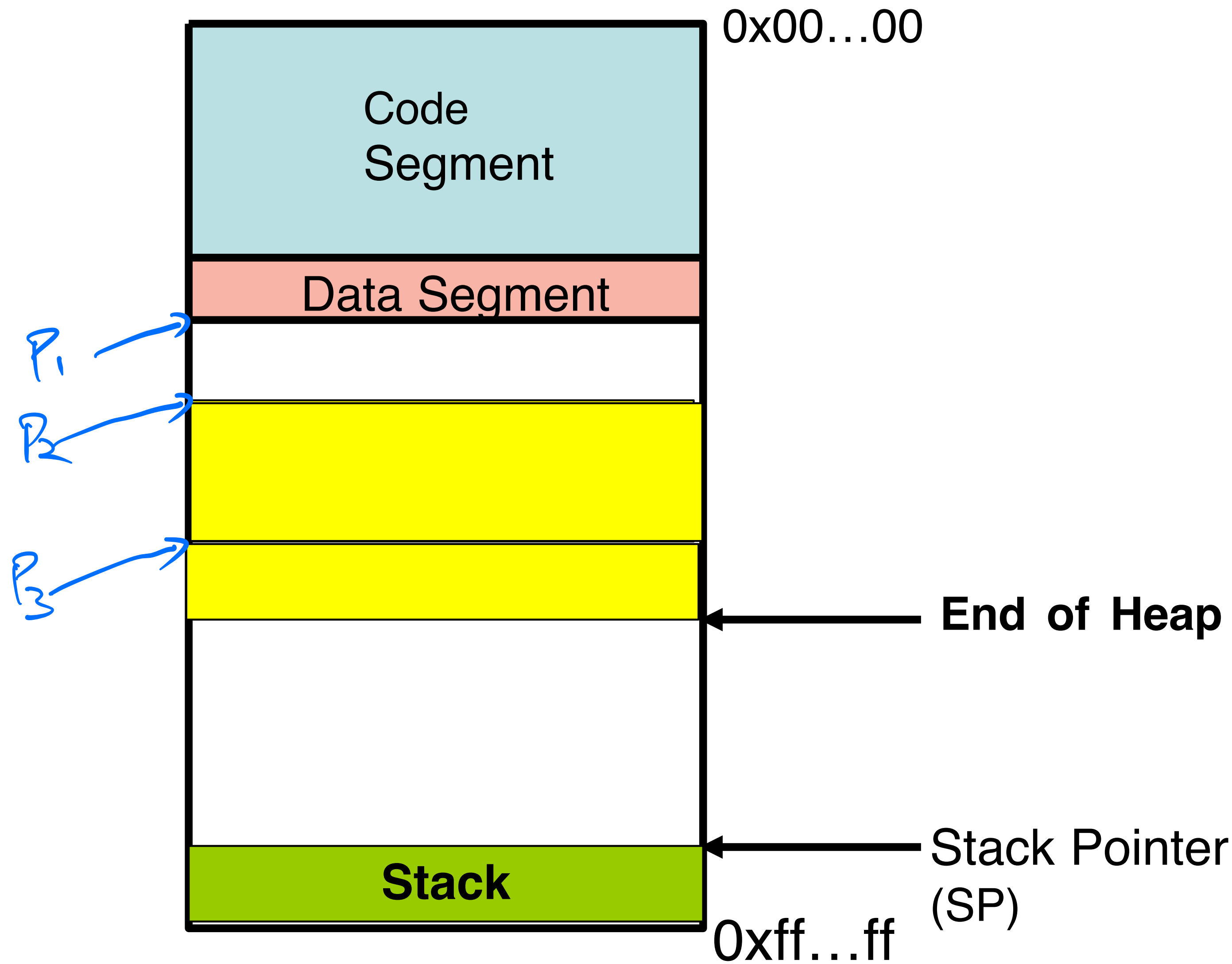
No space in Heap

Syscall

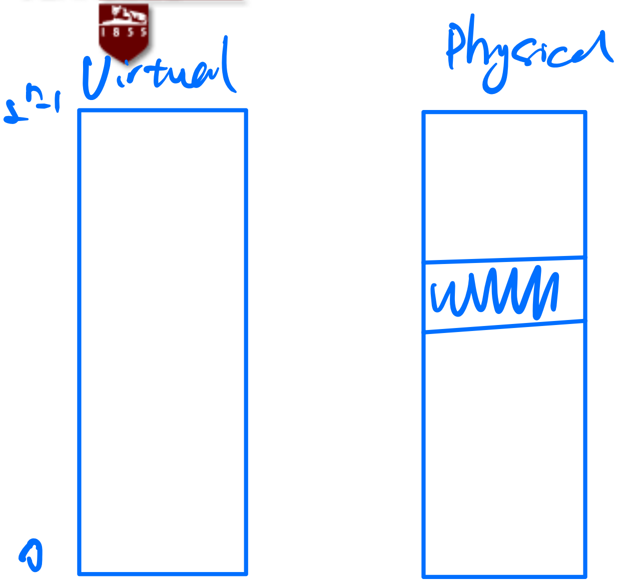
sbrk()
(moves end of Heap ptr)

Malloc()

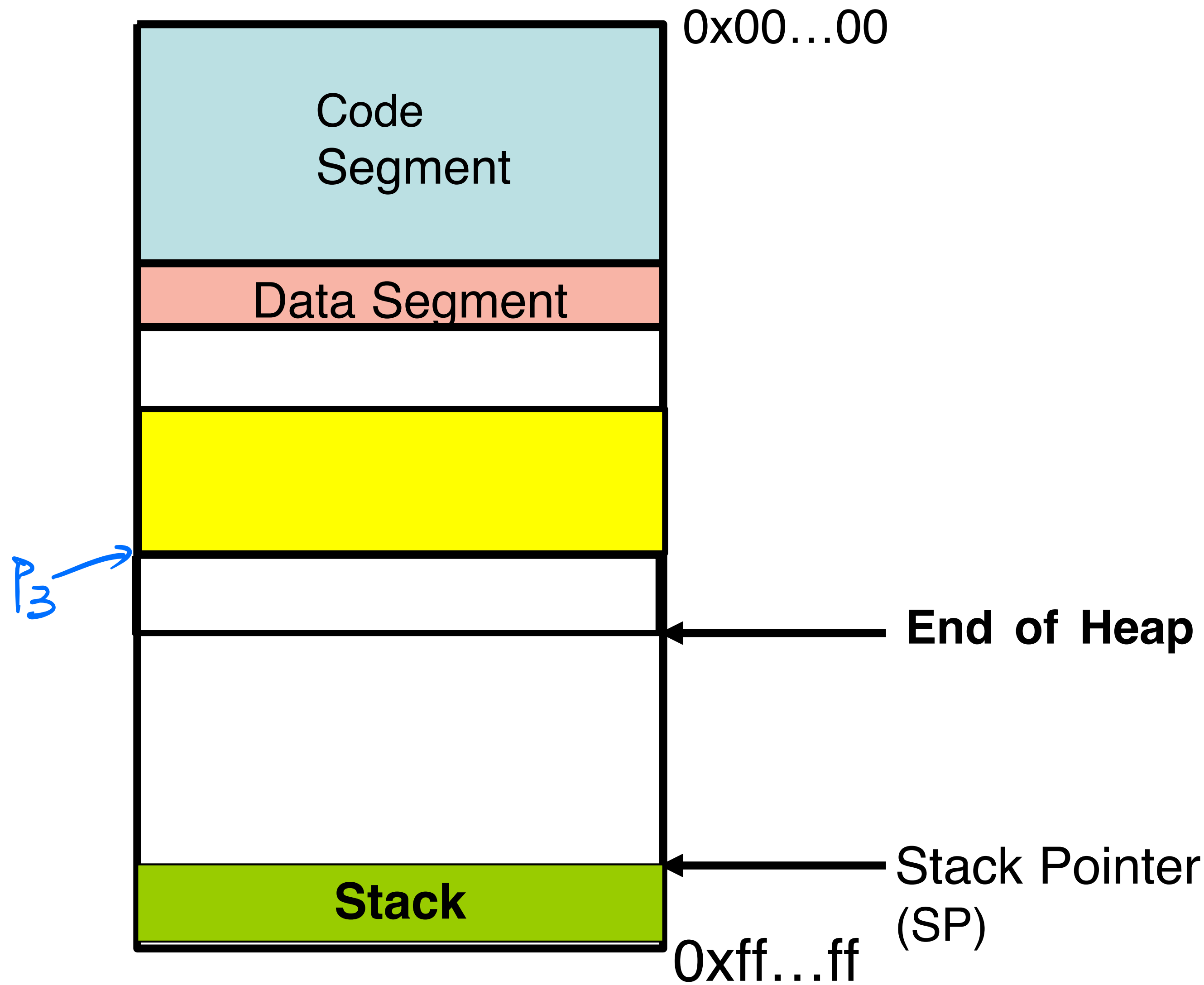
free(p1);



Malloc()



free(p3);



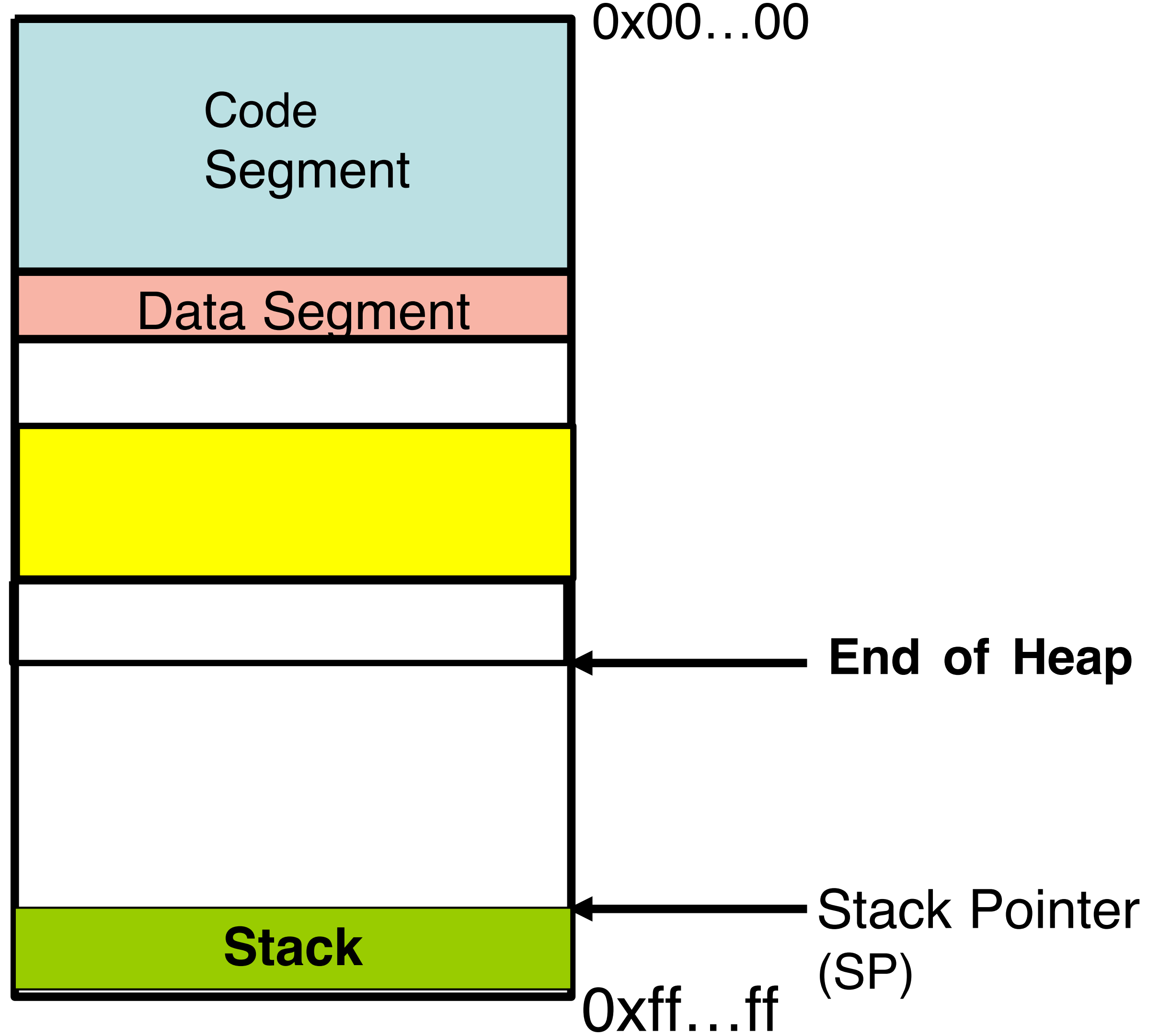
Malloc()

Malloc(150)

What should we do?

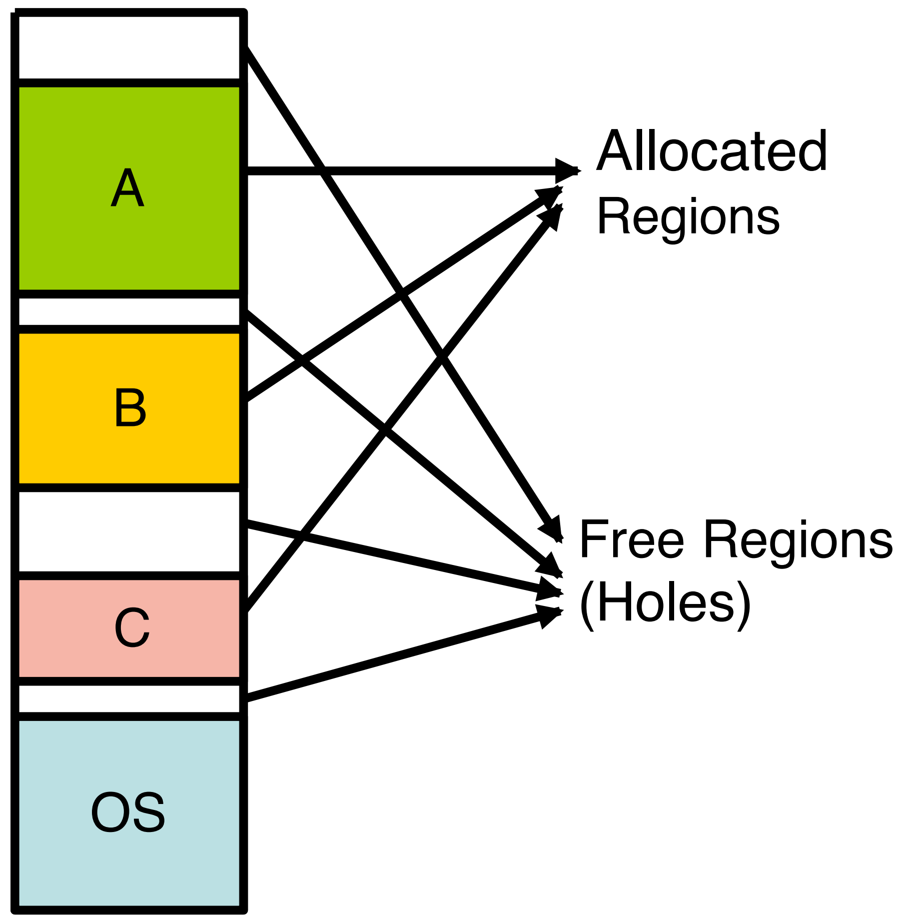
- **Extend heap again (syscall overheads)**
- **Compact (costly)**

frame
segfault



Makes allocation of free space to malloc an important problem

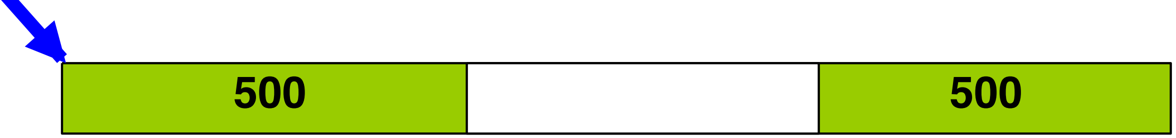
The Memory Allocation Problem



Question: How do we perform these allocations?

Goals

- Program wants allocation to be contiguous. Note "p = malloc(1000)" requires 1000 bytes to be contiguous for p[842] to work



- Allocation() and Free() should be fairly efficient

- Should be able to satisfy more requests at any time (i.e., the sum total of holes should be close to 0 with waiting requests).

Solution strategies

- Contiguous allocation
  - The requested size is granted as 1 big contiguous chunk.
  - E.g., first-fit, best-fit, worst-fit, buddy-system – Algorithm
- Non-contiguous physically but virtually contiguous
  - The requested size is granted as several pieces (and typically each of these pieces is of the same – fixed - size).
    - E.g., paging
- Use the former to do virtually contiguous in latter

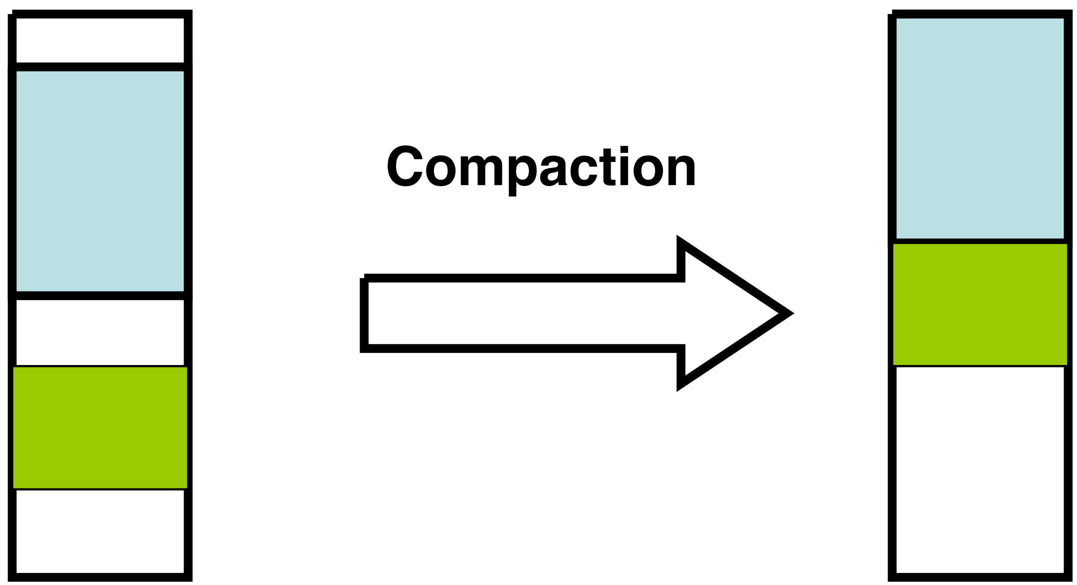
Contiguous allocation

- **Data structures:**
  - **List of allocated regions**
  - **List of holes.**
- **Find a hole and make the allocation (and it may result in a smaller hole).**
- **Eventually, you may get a lot of holes that become small enough that they cannot be allocated individually.**
- **This is called external fragmentation.**

external

fragmentation

Easing external fragmentation



Note that this can be done only with relocatable code and data (use indirect/indexed/relative addressing)

But compaction is expensive and we want to do this as infrequently as possible.

Contiguous allocation

- **Which hole to allocate for a given request?**
- **First-fit**
  - **Search through the list of holes. Pick the first one that is large enough to accommodate this request.**
  - **Though allocation may be easy, it may not be very efficient in terms of fragmentation.**

- Best Fit
  - Search through the entire list to find the smallest hole that can accommodate the given request.
  - Requires searching through the entire list (or keeping it in sorted order).
  - This can actually result in very small sized holes making it undesirable.

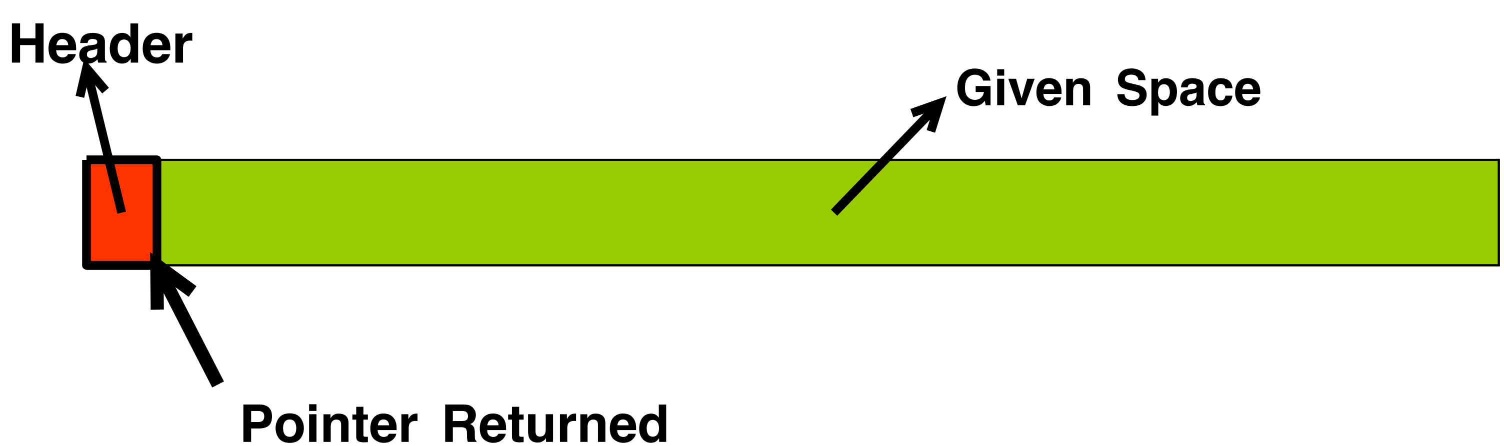
- **Worst fit**
  - Pick the largest hole and break that.
  - The goal is to keep the size of holes as large as possible.
  - Allocation is again quite expensive (searching through entire list or keeping it sorted).

What do you do for a free(p)?

- You need to determine whether nearby regions (on either side) is free, and if so you need to make a larger hole.
- To do this:
  - We need to know what size you are free -ing
  - We need to know whether either side of this is free

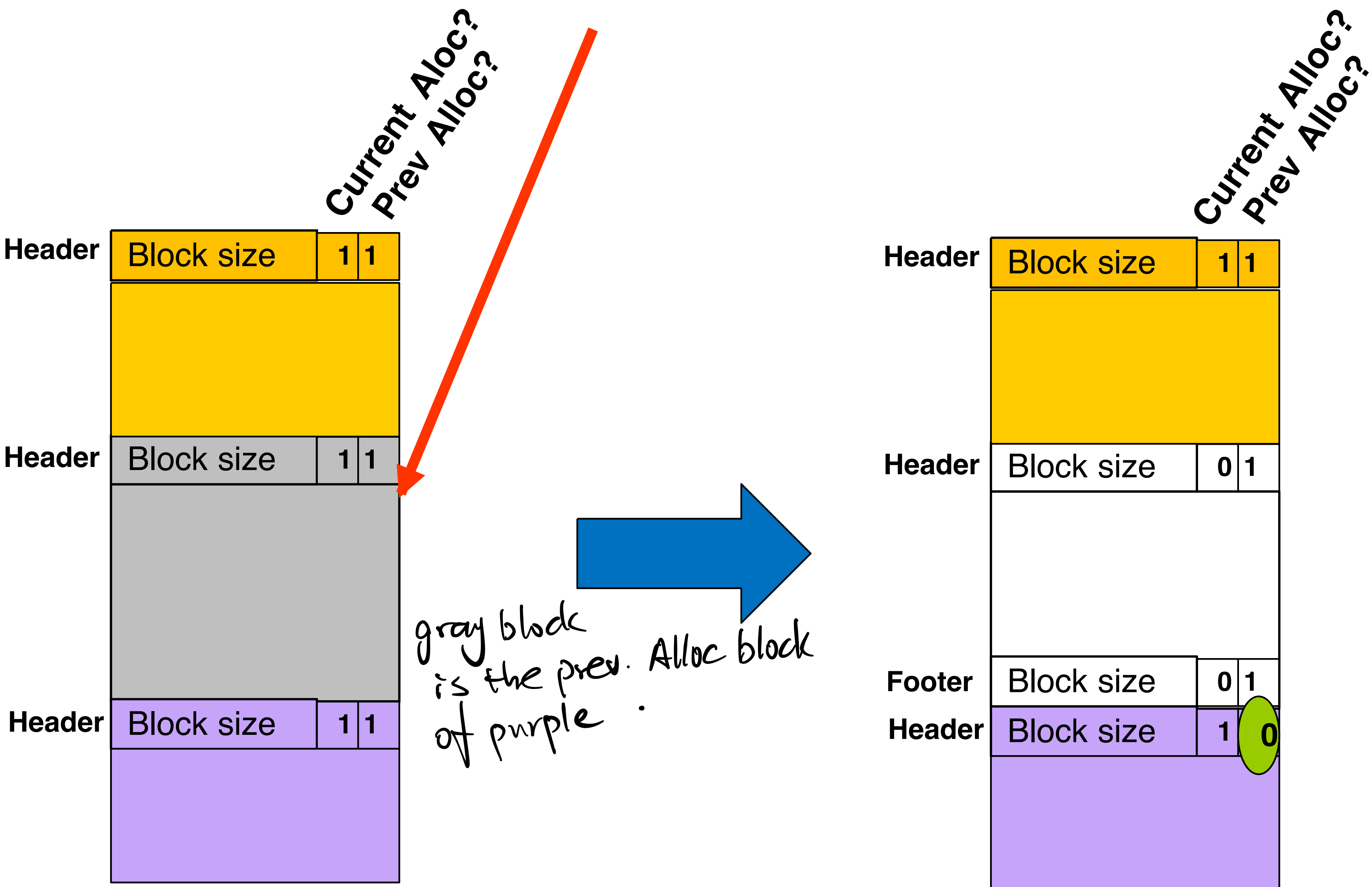
How do you know what size you are freeing?

- Allocate extra space for a small header and return pointer after this header for an allocation.
- Do a negative offset of the pointer passed by free to find size!



How do you know whether either side of what you are free-ing are free, and if so merge with them to create a larger hole?

free(p) – Case 1



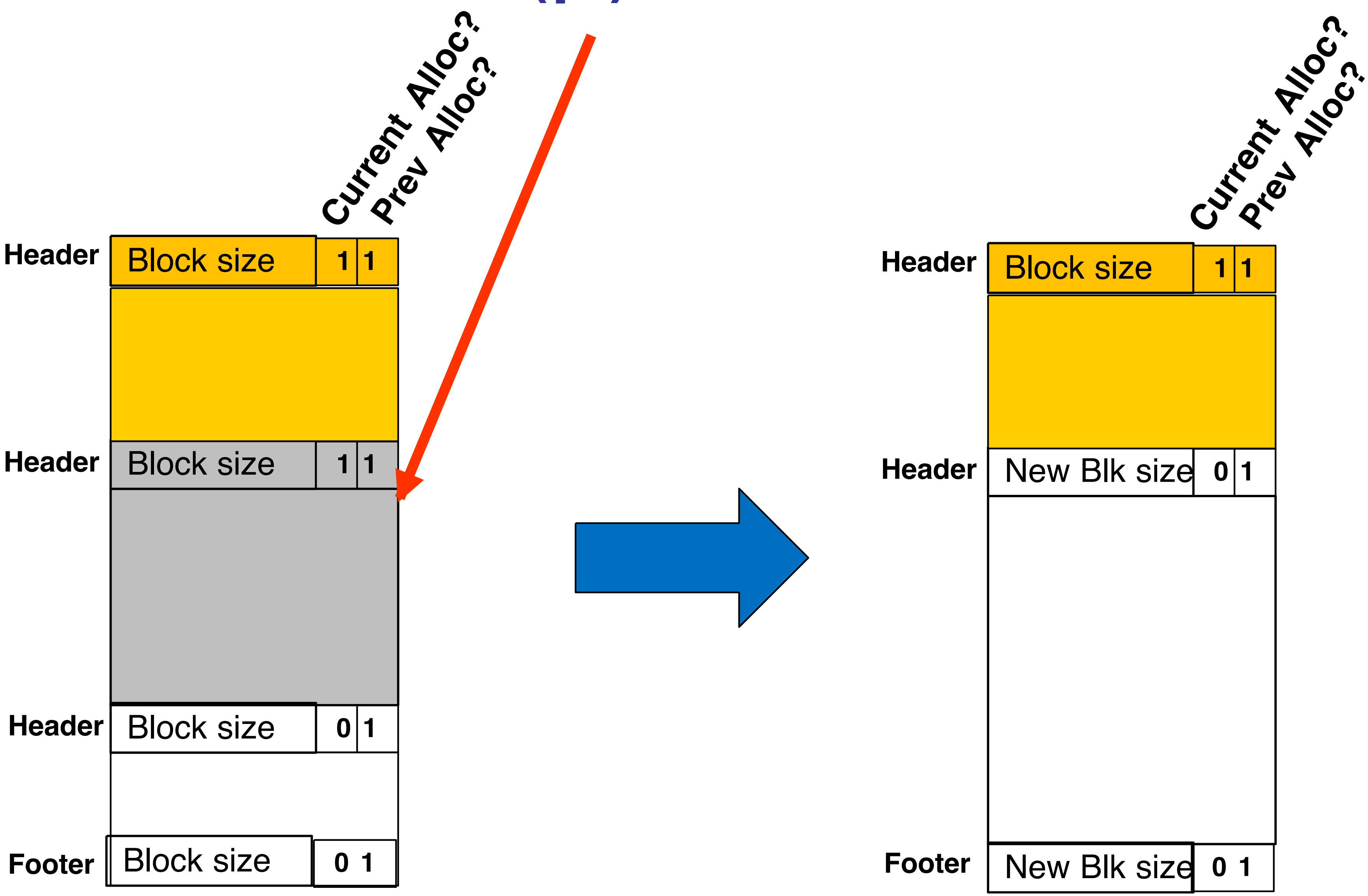
gray block is the prev. Alloc block of purple.

free(ptr)

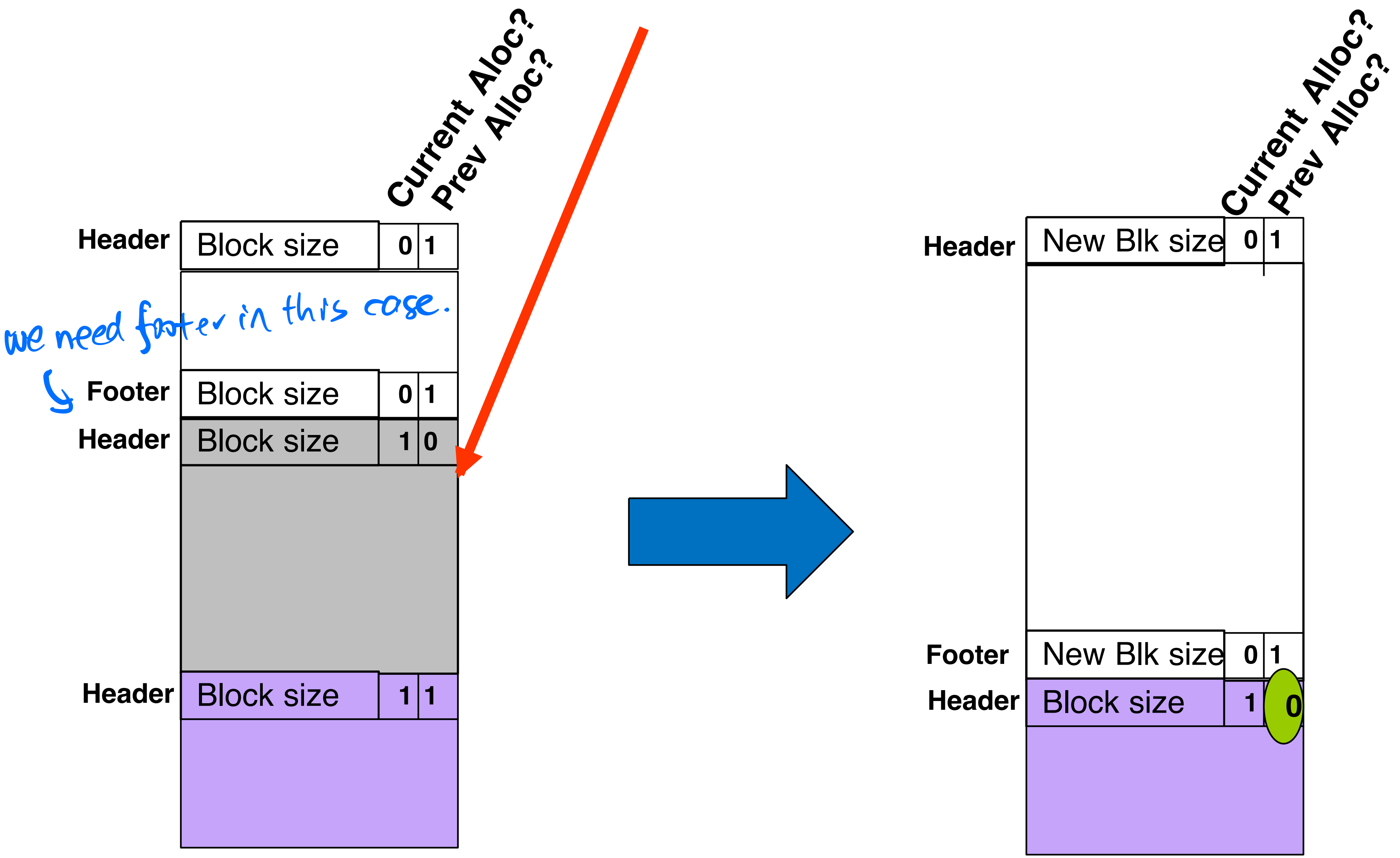
header_next = ptr + header

→ blk-size + size of (footer type)

free(p) – Case 2

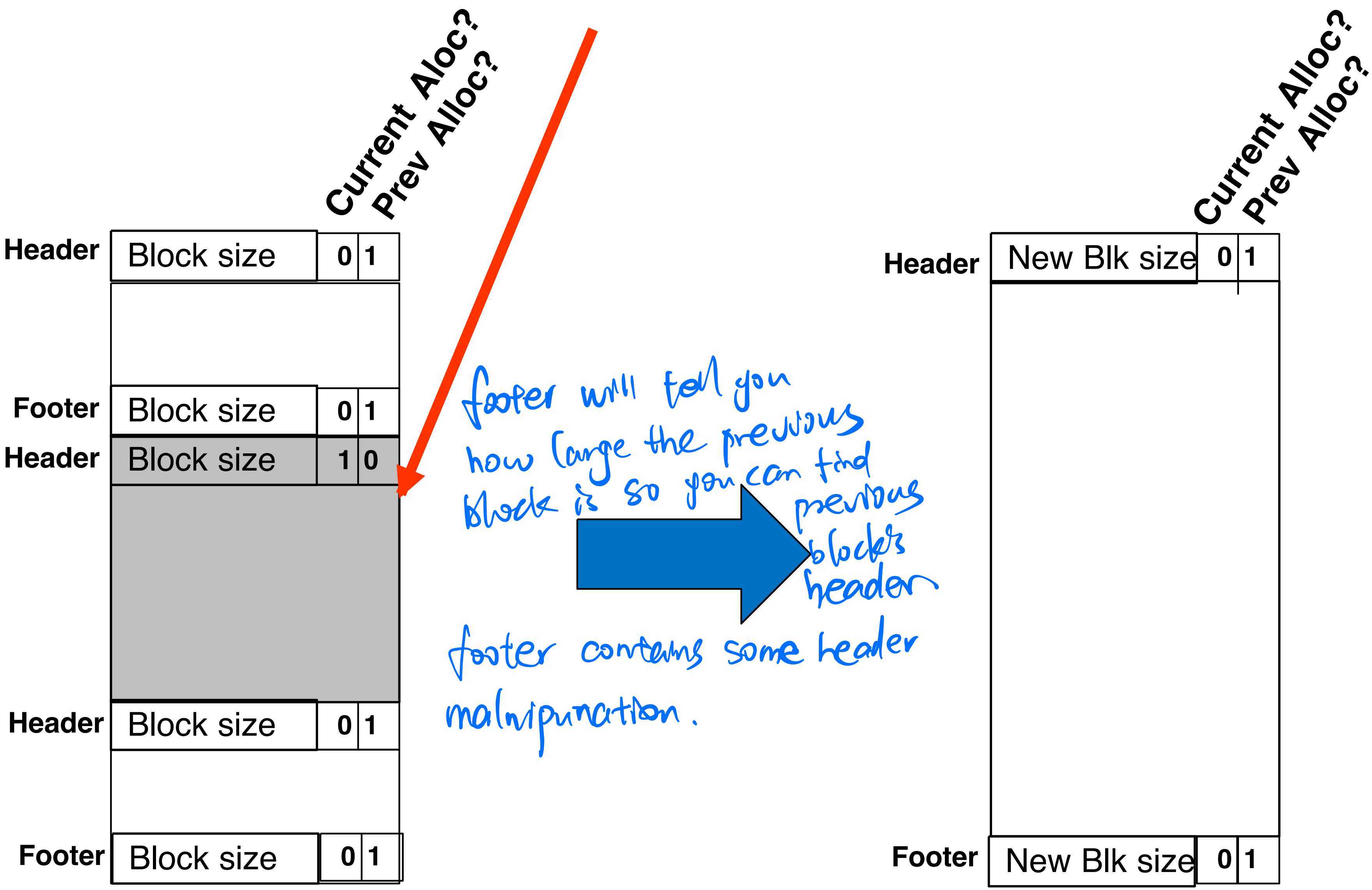

Current Alloc?
Prev Alloc?
Header
Block size
1
1
Header
Block size
1
1
Header
Block size
0
1
Footer
Block size
0 1
Current Alloc?
Prev Alloc?
Header
Block size
1
1
Header
New Blk size
0
1
Footer
New Blk size
0 1

free(p) – Case 3



Current Aloc?
Prev Alloc?
Header
Block size
0
1
we need footer in this case.
Footer
Block size
0
1
Header
Block size
1
0
Header
Block size
1
1
Current Alloc?
Prev Alloc?
Header
New Blk size
0
1
Footer
New Blk size
0
1
Header
Block size
1
0

free(p) – Case 4



footer will tell you how large the previous block is so you can find previous blocks header

footer contains same header malnipunation.

PENNSTATE

Costs

- Free: Look only at neighbors (O(1))
- Allocation: Search through list (O(N)).

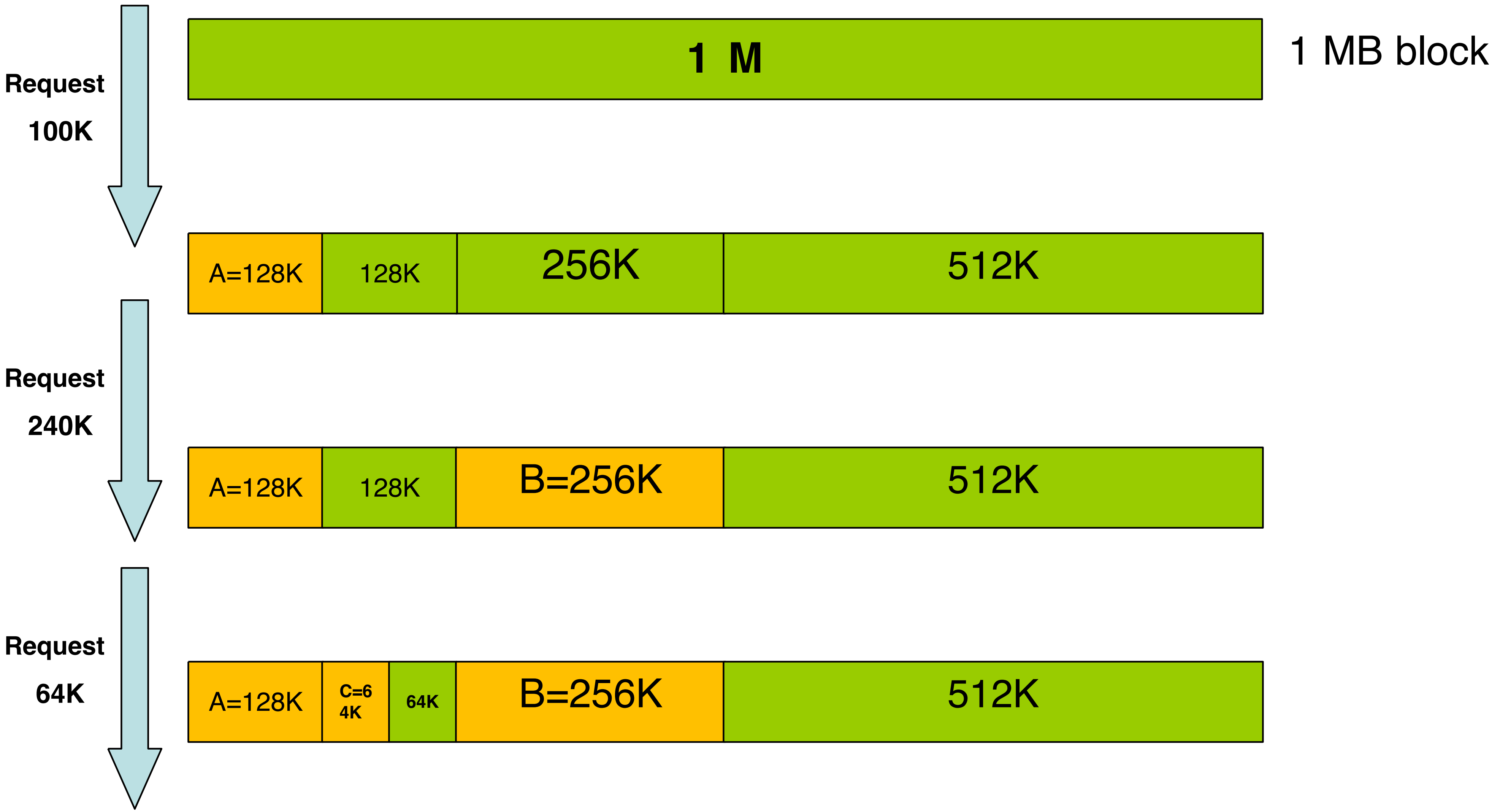
Can we do better?

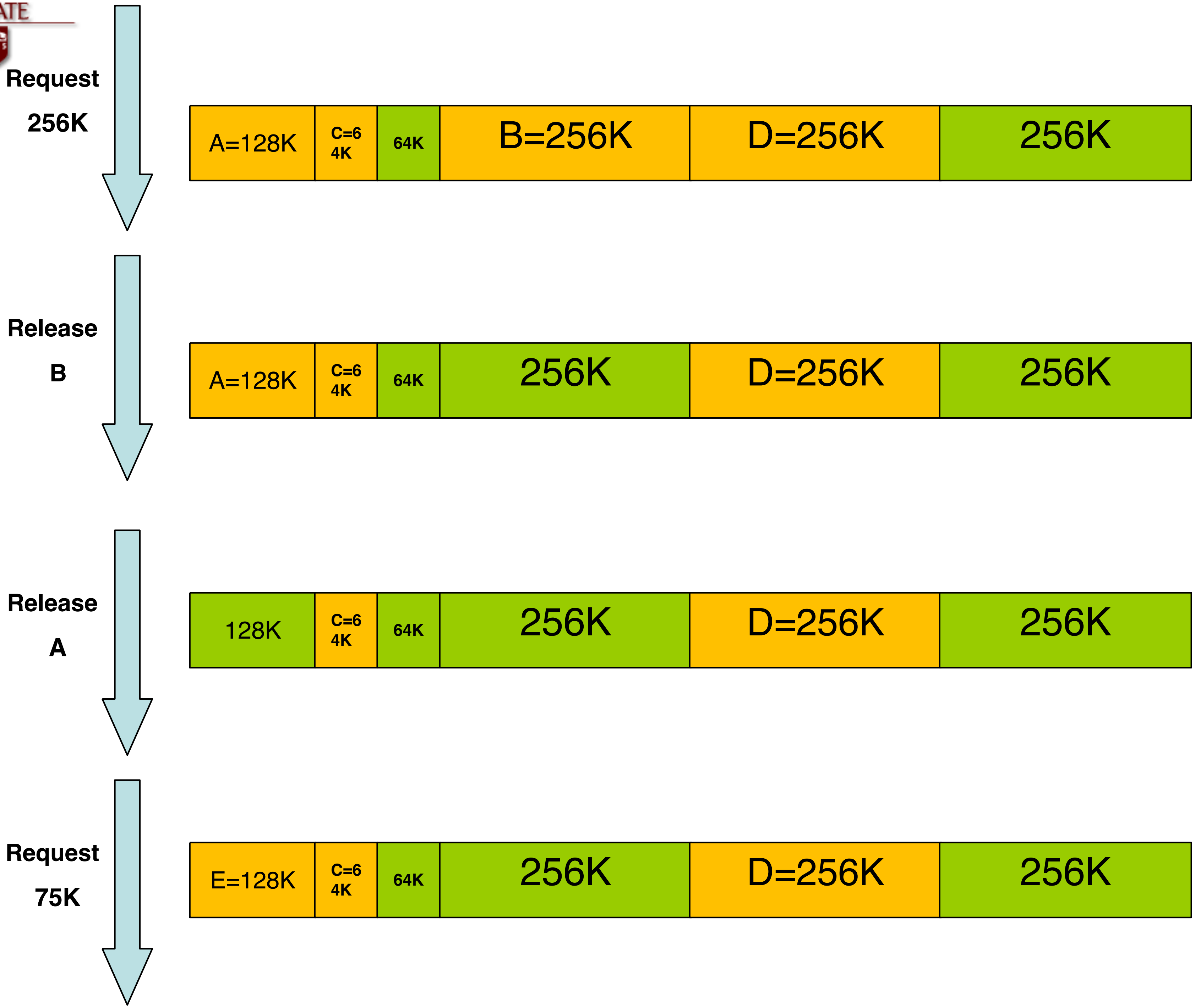
Buddy System

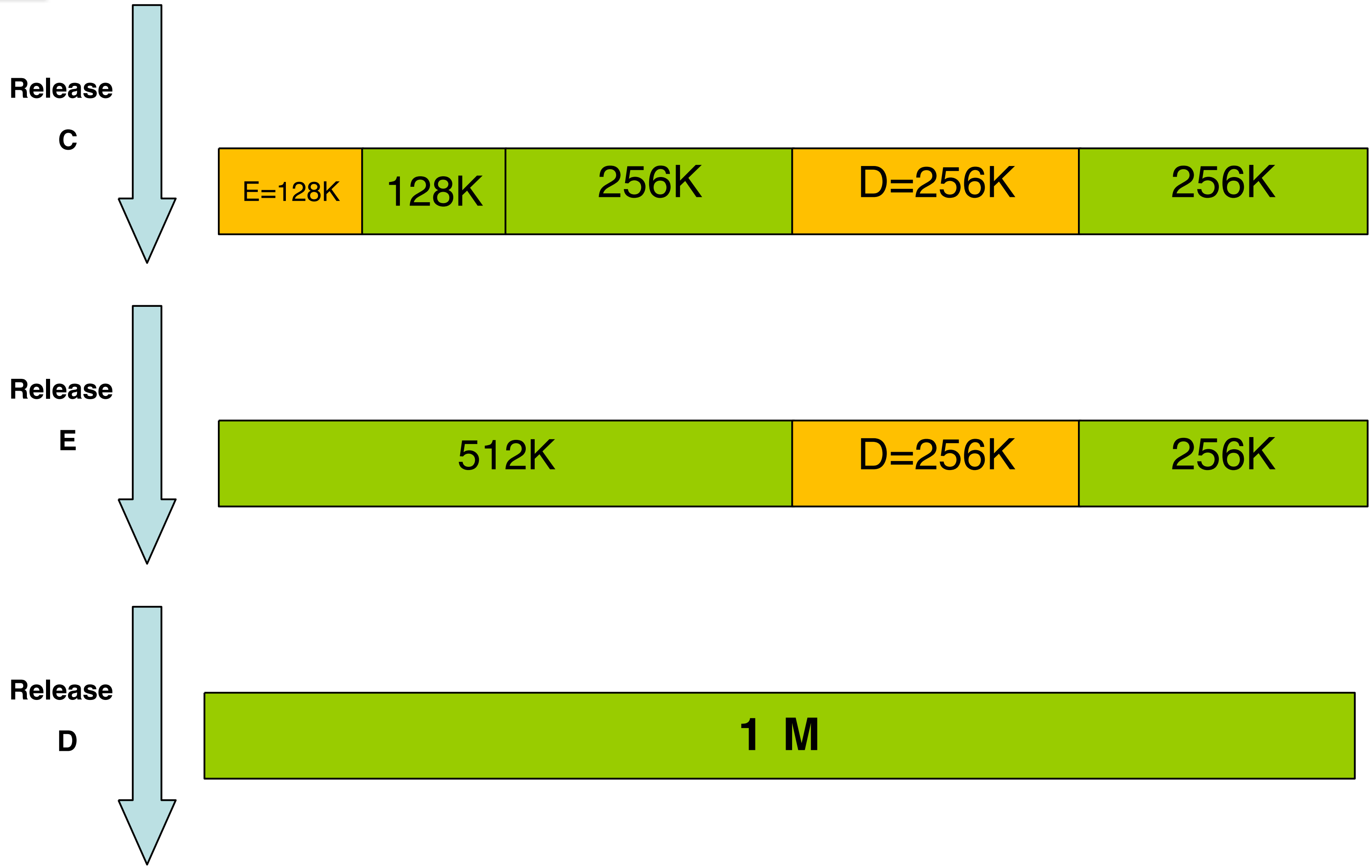
- Log(N) cost for allocation/free.

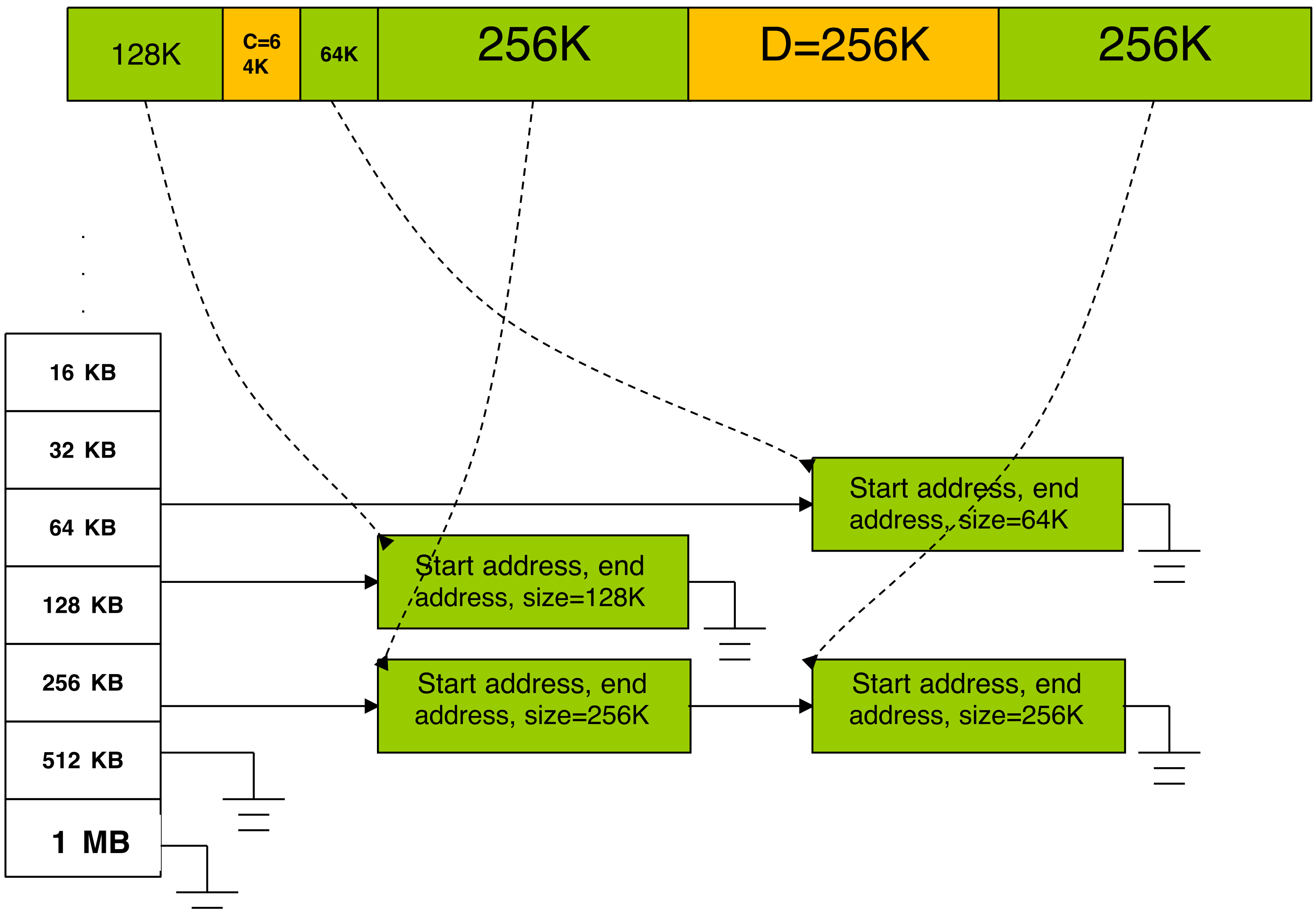
- Round up request to next power of 2, and find a block to allocate

An Example


1 M
1 MB block
Request
100K
A=128K
128K
256K
512K
Request
240K
A=128K
128K
B=256K
512K
Request
64K
A=128K
C=6
4K
64K
B=256K
512K


Request
256K
A=128K
C=64K
64K
B=256K
D=256K
256K
Release
B
A=128K
C=64K
64K
256K
D=256K
256K
Release
A
128K
C=64K
64K
256K
D=256K
256K
Request
75K
E=128K
C=64K
64K
256K
D=256K
256K

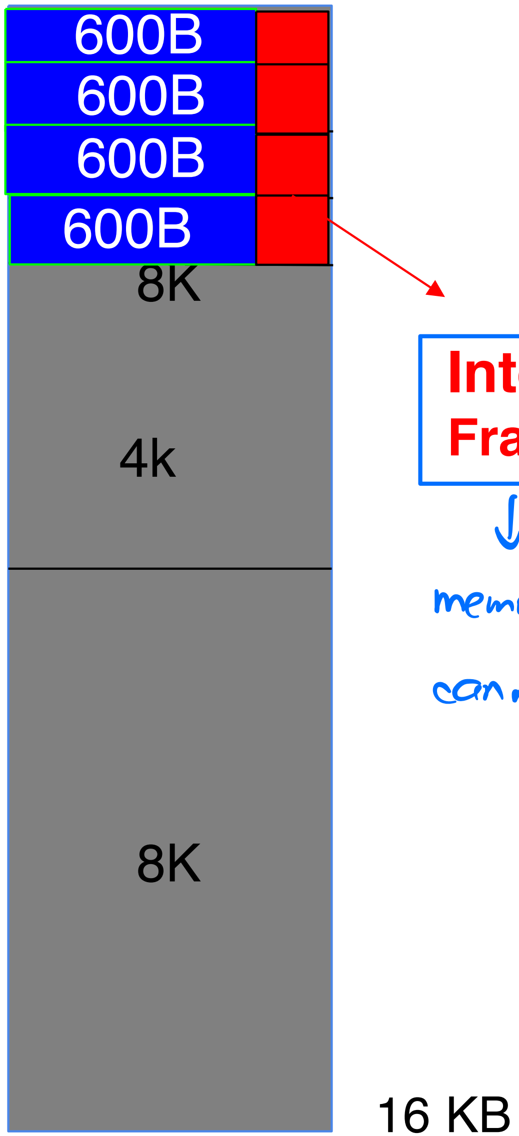

Release
C
E=128K
128K
256K
D=256K
256K
Release
E
512K
D=256K
256K
Release
D
1 M



List of Available Holes

Problems with Buddy

– might waste memory



Internal Fragmentation!

memory space that cannot access

Object *objs[4];

for (i=0; i < 4; i++){

obj[i] = malloc(600)

}

***Object* size is 600 Bytes.**

The problem can worsen for larger sizes because of rounding off to next power of 2 !

This is called Internal Fragmentation!

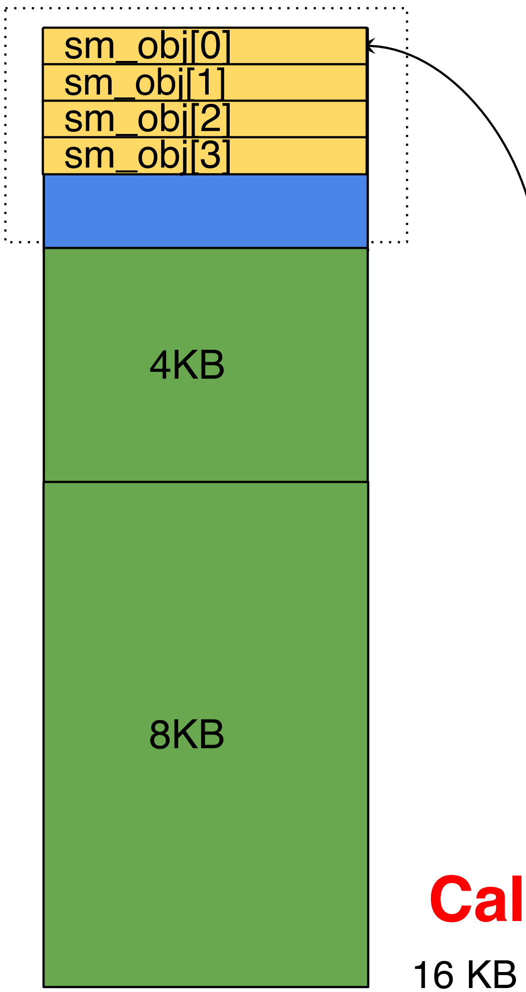
- **Can we leverage such repetitive behavior?**

- **Can we pack more objects into contiguous chunks to avoid internal fragmentation?**

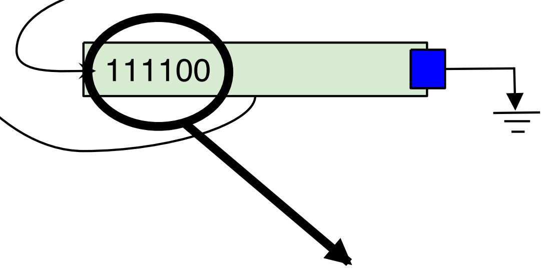
Slab Allocator

- Do continuous allocation of large chunks (e.g. 4K, 8K, …) – called Slabs – using *Buddy* allocator.
- Each Slab will contain objects of same size (e.g. 600 bytes in this example) – Slab type
- An object of a particular size will try to get allocated in the slab of that type/size.
- Slab size is the number of objects of that type/size that this slab can hold.
- When you try to allocate more objects than what a single slab can hold, you allocate one more slab for that type (using Buddy).

Slab Allocator



Slab Descriptor Table				
Type	Size	#Obj	#Used	Slab-ptr
600	4K	6	4	

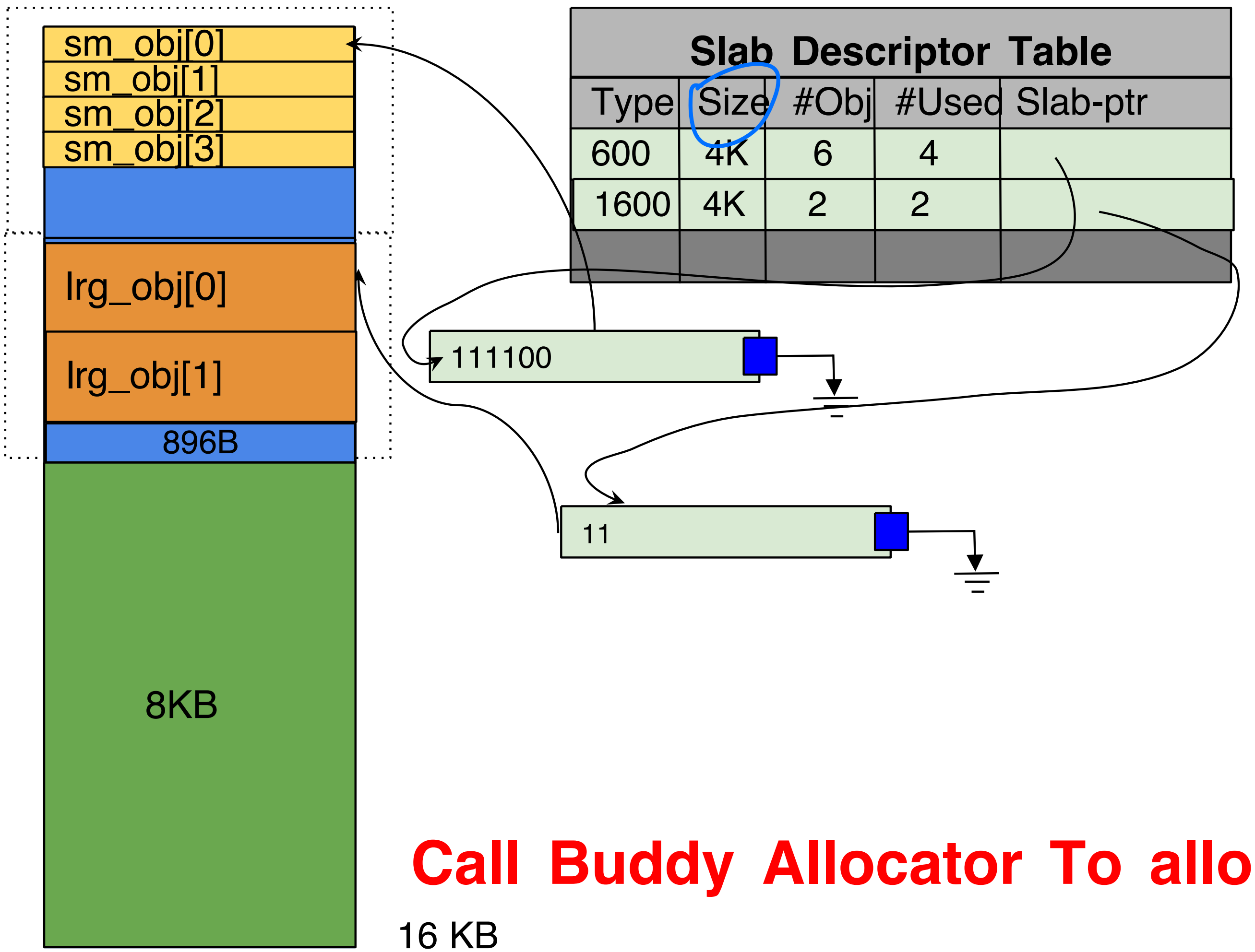


SmallObject* sm_objs[4];
for (int i=0; i < 4; i++) {
sm_objs[i] = malloc(600);
}

Bitmap of what is allocated and what is free in this slab.

Call Buddy Allocator To allocate new Slab

Slab Allocator



Slab Descriptor Table

Type	Size	#Obj	#Used	Slab-ptr
600	4K	6	4	
1600	4K	2	2	

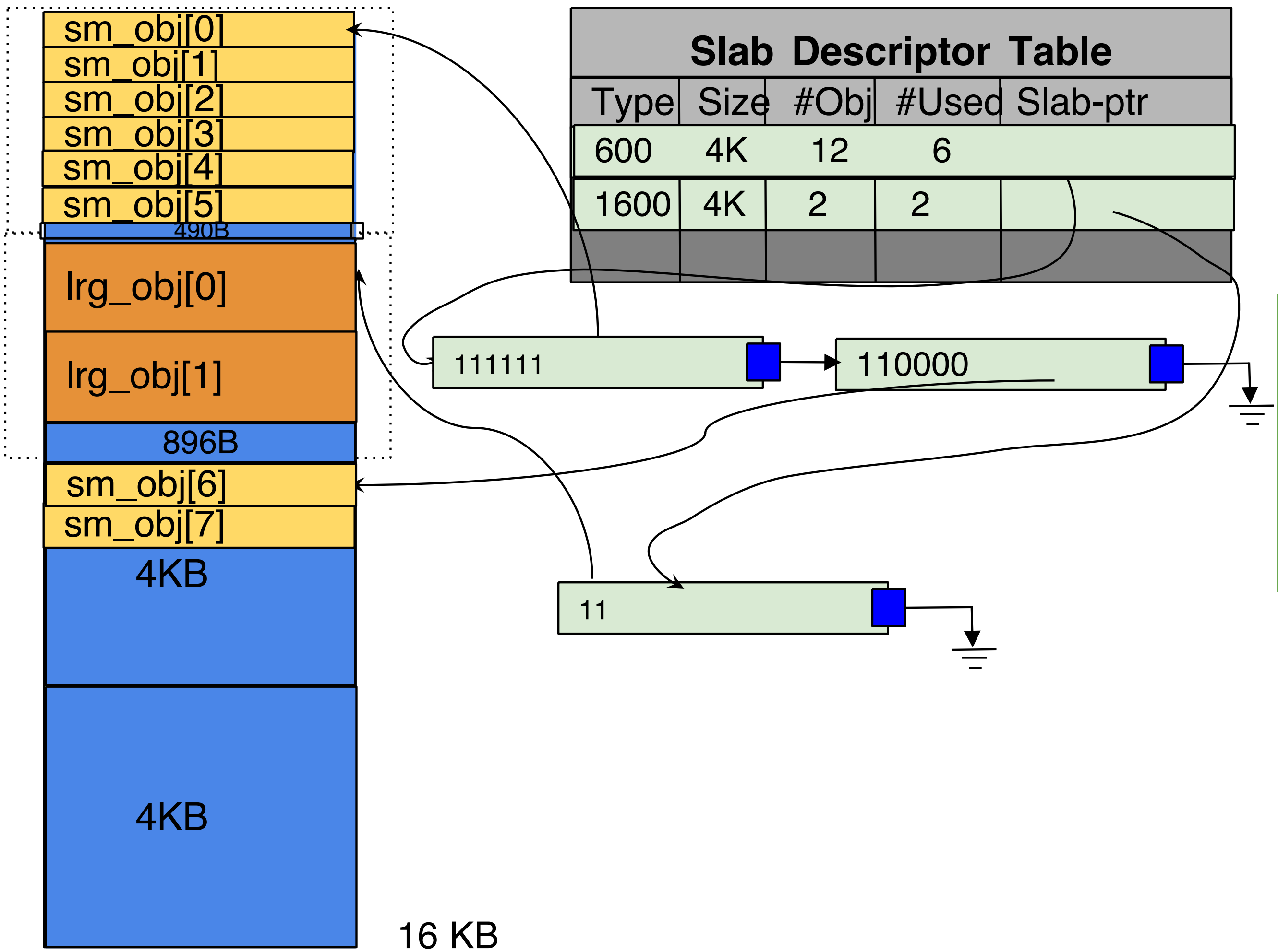
```
SmallObject* sm_objs[4];
for (int i=0; i < 4; i++) {
  sm_objs[i] = malloc(600);
}
```

Larger object

```
LargeObject* lrg_objs[2];
for (int i=0; i < 2; i++) {
 lrg_objs[i] =malloc(1600);
}
```

Call Buddy Allocator To allocate new Slab

Slab Allocator




```
SmallObject* sm_objs[4];
for (int i=0; i < 4; i++) {
  sm_objs[i] = malloc(600);
}
```

```
LargeObject* lrg_objs[2];
for (int i=0; i < 2; i++) {
  lrg_objs[i] =malloc(1600);
}
```

More small objects

```
SmallObject* sm_objs[4];
for (int i=0; i < 4; i++) {
  sm_objs[i] = malloc(600);
}
```

Call Buddy Allocator To allocate new Slab

Slab Allocator: Free

- Again use negative offset to find size of what is being freed.
- Use that to find the corresponding entry in Slab Descriptor Table.
- Go through one slab after another till you find where the chunk to be freed lies.
- Reset the “bit corresponding to that chunk” in the bitmap field.
- A future allocation of the same type can use this freed chunk