CS33 DISCUSSION 8

Brandon Wu

12.5.14 (week 9)

Topics

- Exceptional Control Flow (Chapter 8)
- Virtual Memory and Dynamic Memory (Chapter 9)
  - Including project stuff!

- According to lecture announcement, Linking is not covered on Final exam

Process Control

- A **<u>Process</u>** is a running instance of a program
- Many processes may run concurrently
- May have duplicate instances of the same program running concurrently

Processes can spawn other processes...

Unix Fork( ) system call

- Unix system call for creating new processes
- Process makes a copy of itself
  - Child has exact copy of parent's stack and registers
- Fork returns the process id (pid) of the spawned child
- But if they are exact copies, how can we tell them apart?

Fork example

```
int main() {
    pid_t pid=0;
    pid = fork( );
    if (pid==0)
        printf(“Hello\n”);
    else printf(“World\n”);
}
```

Fork example

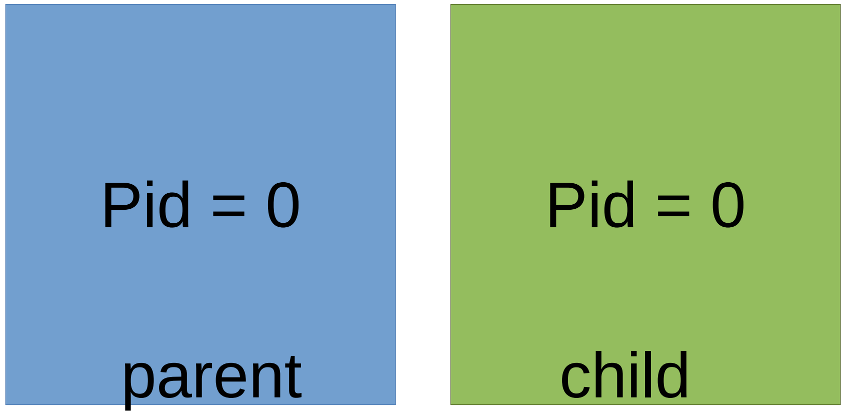
```
int main() {
    pid_t pid=0;
    pid = fork( );
    if (pid==0)
        printf("Hello\n");
    else printf("World\n");
}
```

At time of fork, child gets a copy of parent data

Fork example

```
int main() {
    pid_t pid=0;
    pid = fork( );
    if (pid==0)
        printf(“Hello\n”);
    else printf(“World\n”);
}
```

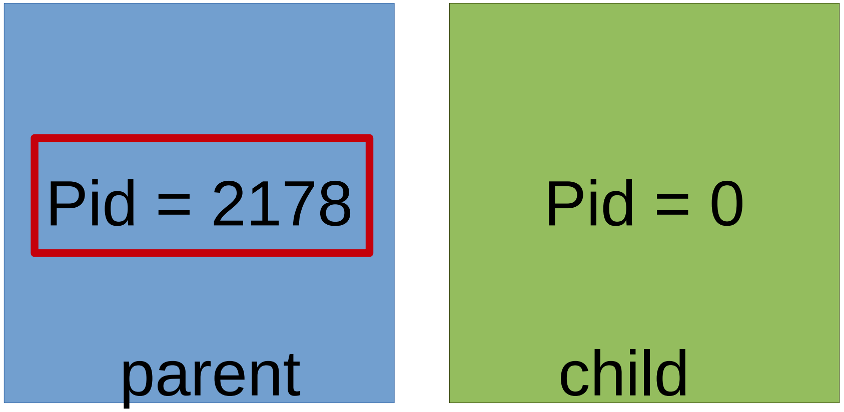
At time of fork, child gets a copy of parent data



Fork example

```
int main() {
    pid_t pid=0;
    pid = fork( );
    if (pid==0)
        printf("Hello\n");
    else printf("World\n");
}
```

Call to fork( ) returns pid of child



Fork example

```
int main() {
    pid_t pid=0;
    pid = fork( );
    if (pid==0)
        printf(“Hello\n”);
    else printf(“World\n”);
}
```

So we use the value of pid to id parent and child

Fork example

```
int main() {
    pid_t pid=0;
    pid = fork( );
    if (pid==0)
        printf("Hello\n");
    else printf("World\n");
}
```

So we use the value of pid to id parent and child

Pid = 2178
>> "World"

parent

Pid = 0
>> "Hello"

child

Example From Lecture

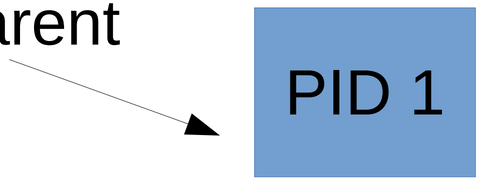
```
int main() {
    fork();
    fork();
    fork();
    printf("hello\n");
    return 0;
}
```

Q: How many times is hello printed?

Example From Lecture

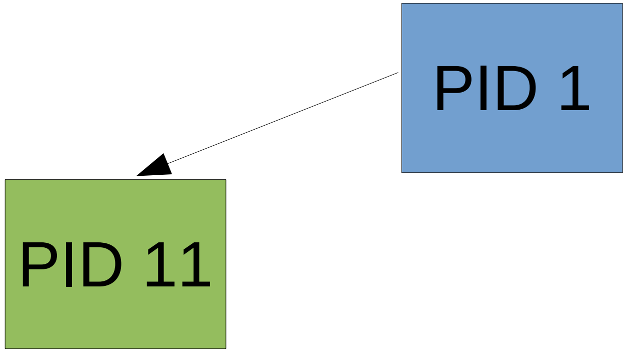
```c
int main() {
    fork();
    fork();
    fork();
    printf("hello\n");
    return 0;
}
```

The parent



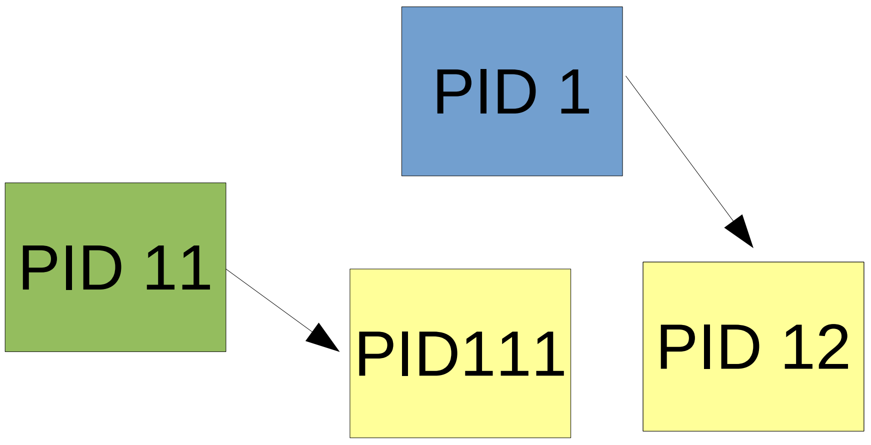
Example From Lecture

```
int main() {
    fork();
    fork();
    fork();
    printf("hello\n");
    return 0;
}
```



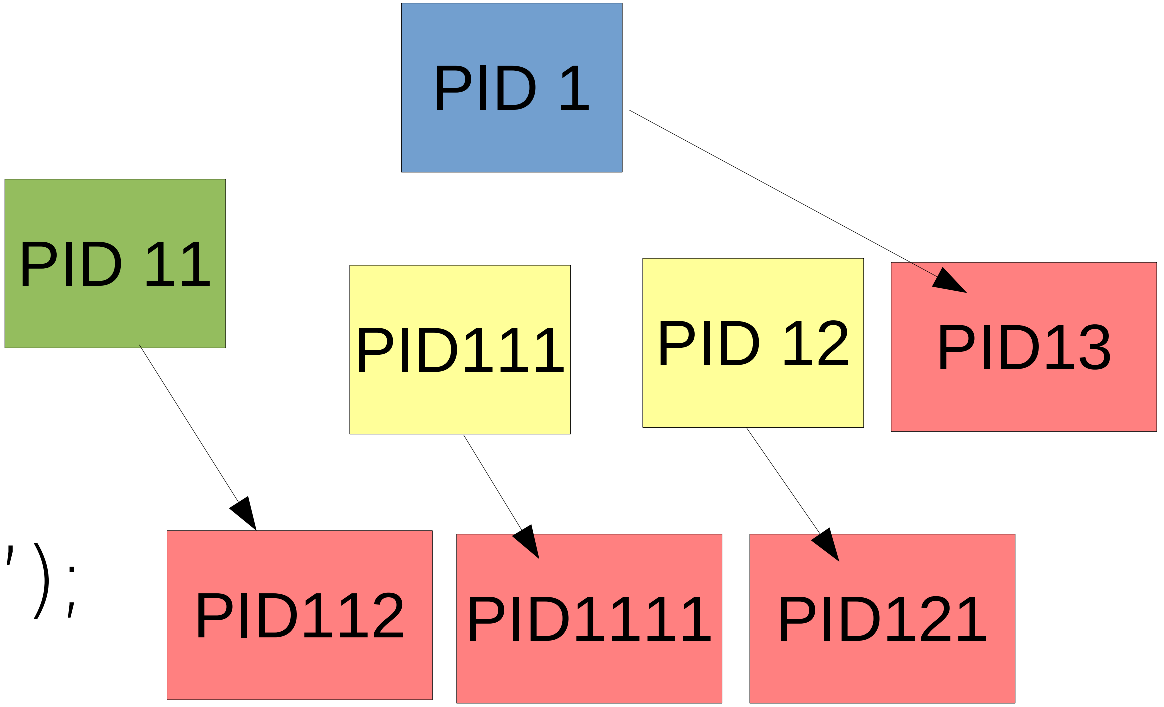
Example From Lecture

```
int main() {
    fork();
    fork();
    fork();
    printf("hello\n");
    return 0;
}
```



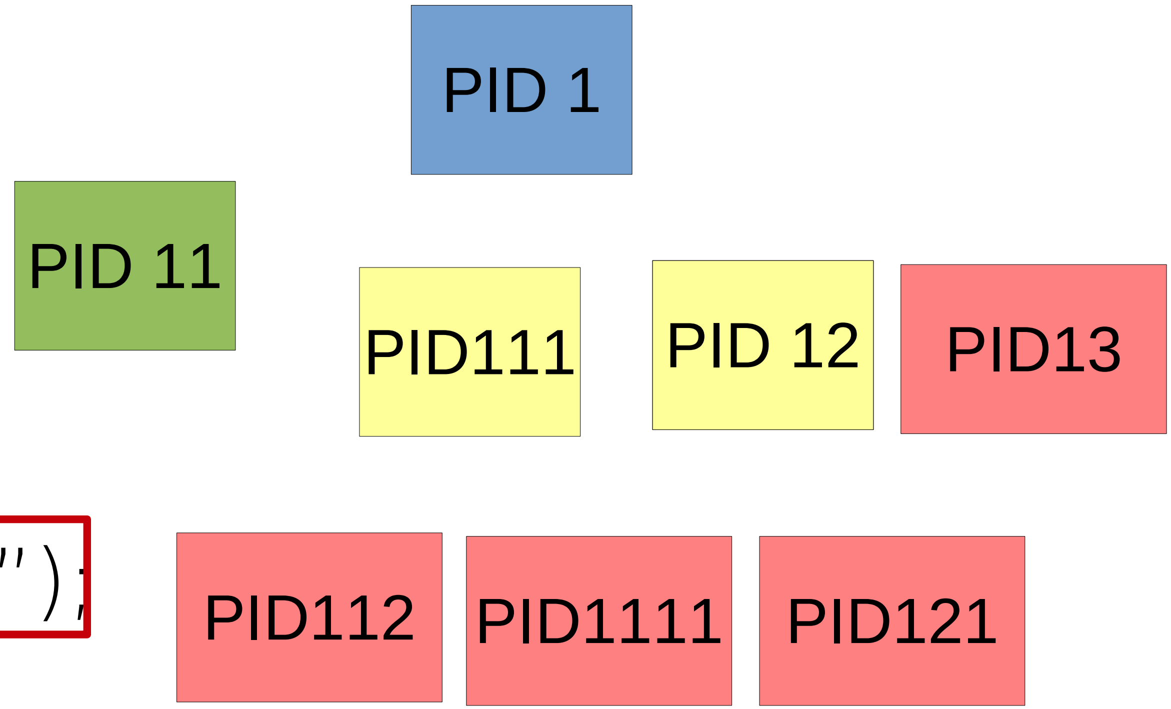
Example From Lecture

```
int main() {
    fork();
    fork();
    fork();
    printf("hello\n");
    return 0;
}
```



Example From Lecture

```
int main() {
    fork();
    fork();
    fork();
    printf("hello\n");
    return 0;
}
```



"hello" prints 8 times

Concurrency

- Processes run concurrently (scheduled by OS)
- No guarantee on sequence of execution
  - i.e. process A may run before or after process B
  - May be interleaved
  - May be different every time you execute

CHAPTER 9:

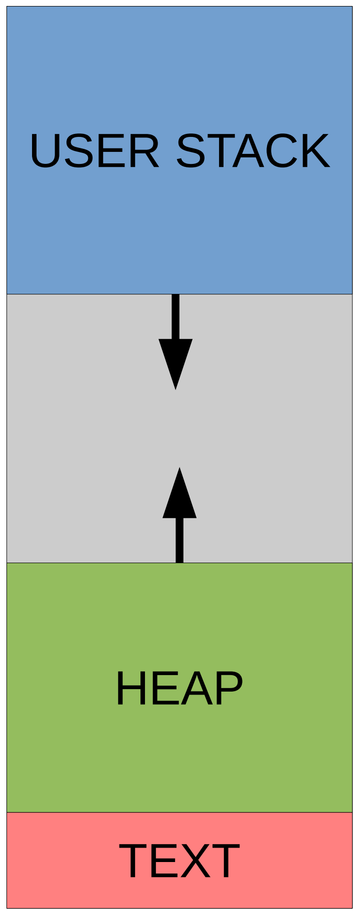
VIRTUAL MEMORY

Motivation

- What is missing from our current memory model?

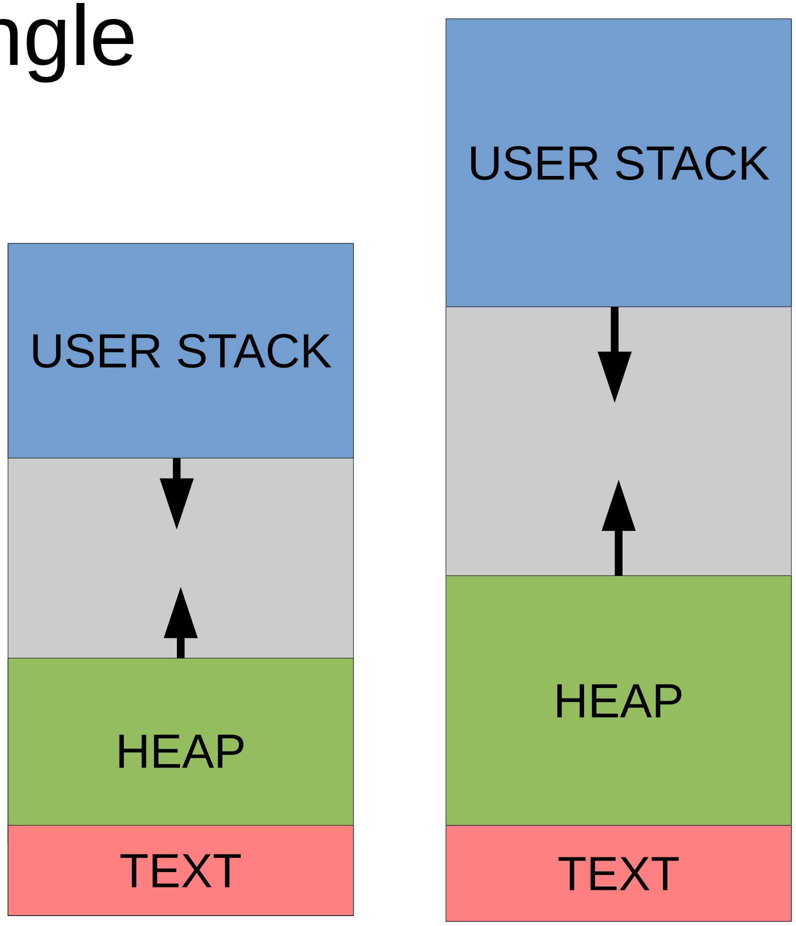
Multiple Processes

- Memory model for single process
- What about multiple processes?



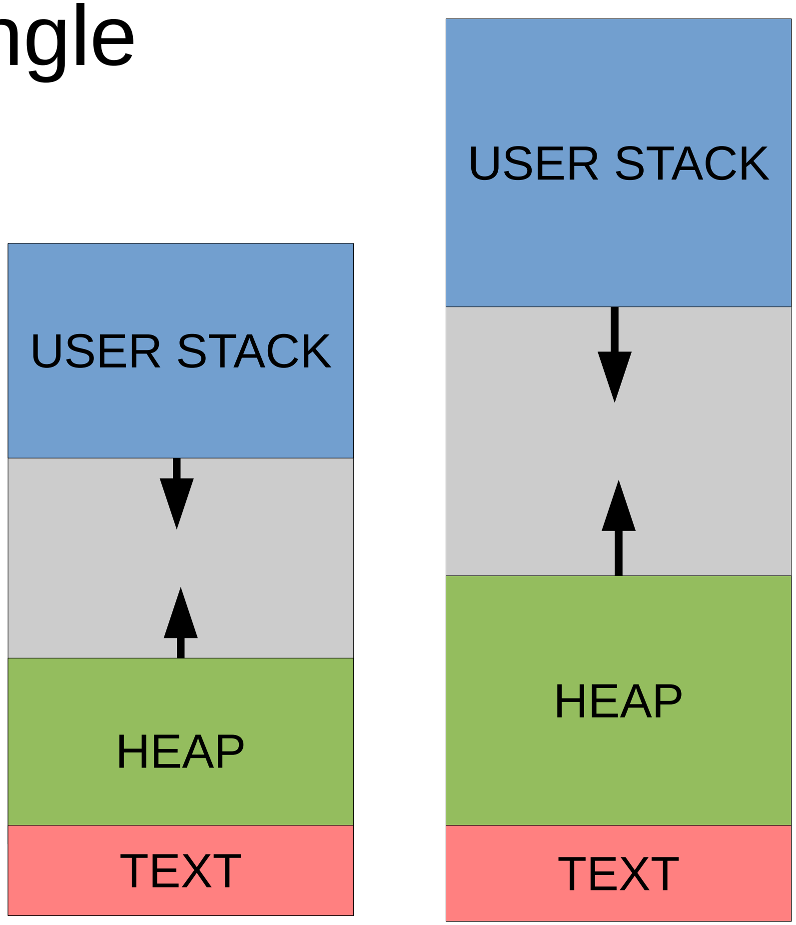
Multiple Processes

- Memory model for single process
- What about multiple processes?
  - Each needs a stack, heap, etc



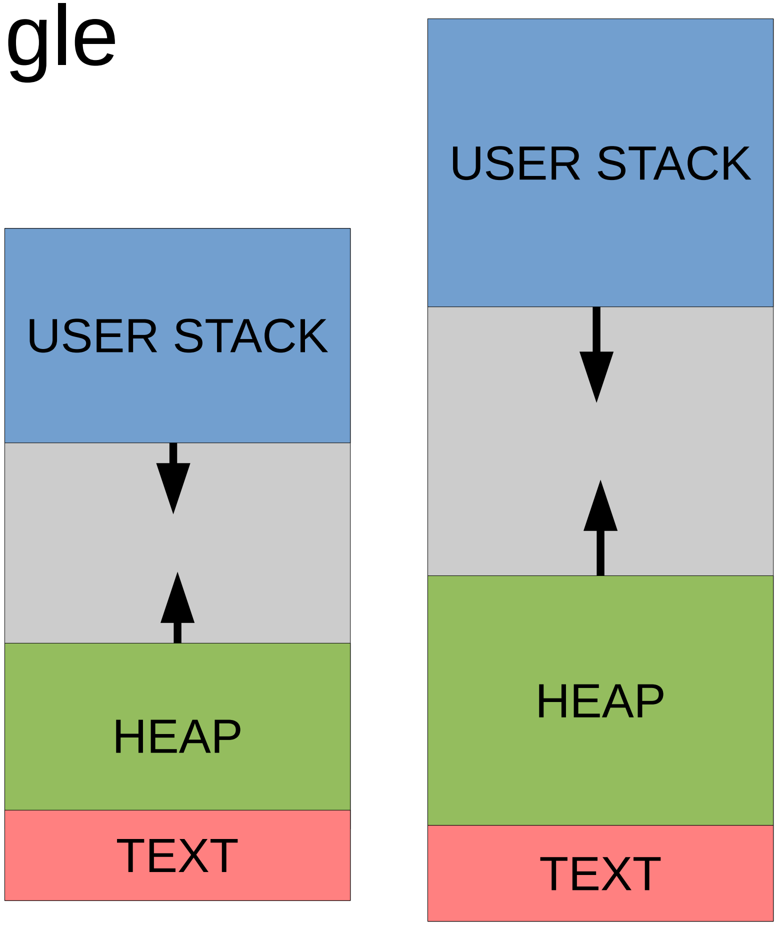
Multiple Processes

- Memory model for single process
- What about multiple processes?
  - Each needs a stack, heap, etc
- Do we share (divide) the memory space?



Multiple Processes

- Memory model for single process
- What about multiple processes?
  - Each needs a stack, heap, etc
- Do we share (divide) the memory space?
  - If so, processes have to know about eachother



Size of Address Space

- For 32 bit address, 4 GB of addressable memory

Size of Address Space

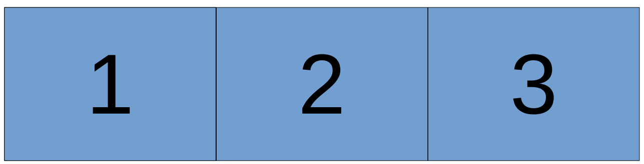
- For 32 bit address, 4 GB of addressable memory
  - What if we have less than 4 GB?
  - Should application developers have to worry about amount of available memory?
- How do we solve these problems?

Virtual Memory

- Every process “sees” full address space with exclusive access
- OS and hardware handles sharing of physical resources amongst running processes
  - And the pieces of memory don't have to be contiguous

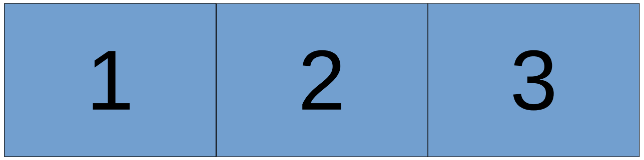
Virtual Memory: Basic Idea

- Process A Virtual Memory



Virtual Memory: Basic Idea

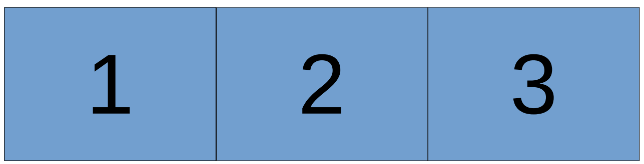
- Process A Virtual Memory



- Process B Virtual Memory

Virtual Memory: Basic Idea

- Process A Virtual Memory

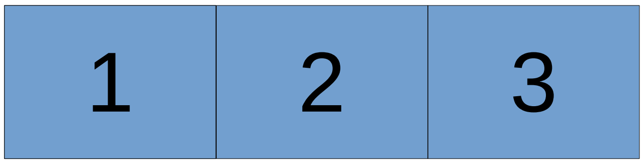


- Process B Virtual Memory

- Main Memory

Virtual Memory: Basic Idea

- Process A Virtual Memory

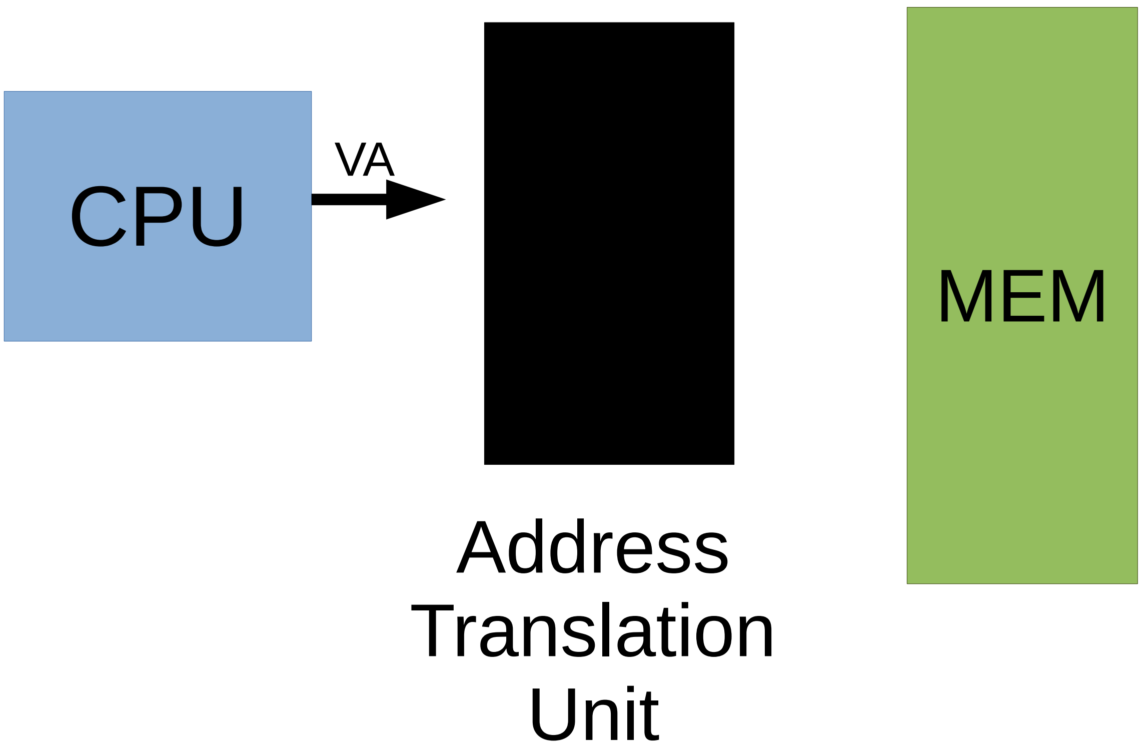


- Process B Virtual Memory

- Main Memory – only store blocks in use

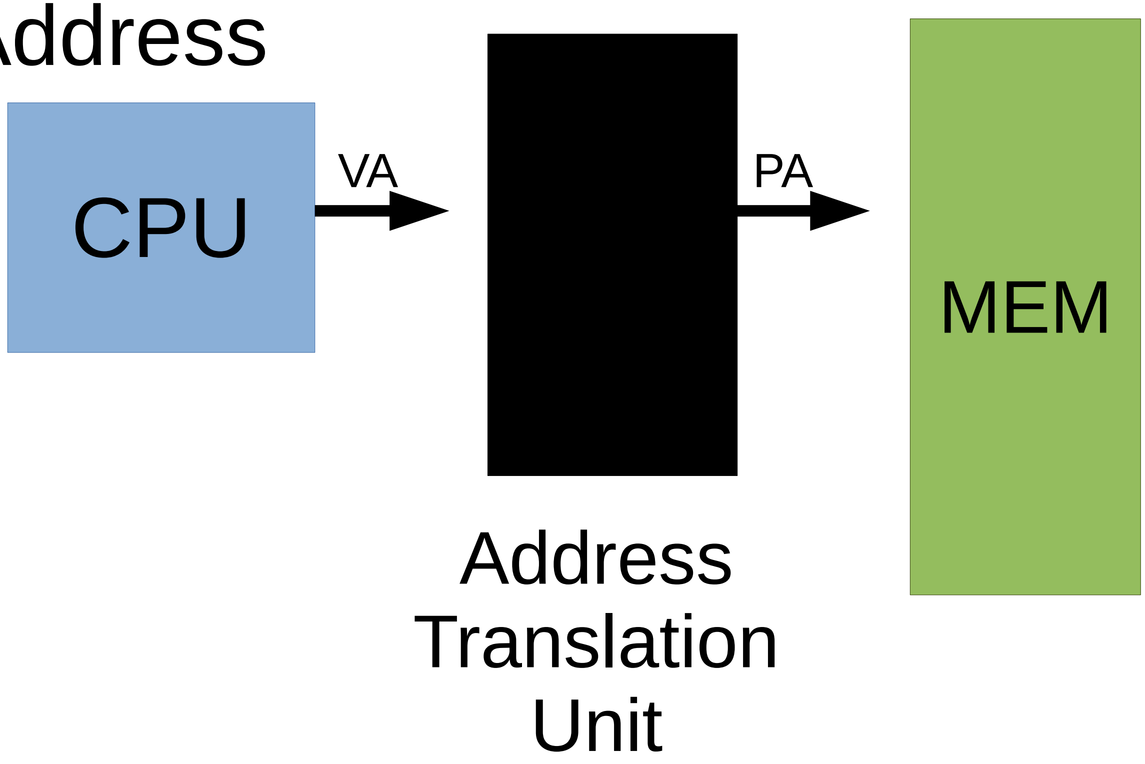
Virtual to Physical Address Translation

- Program issues address instruction



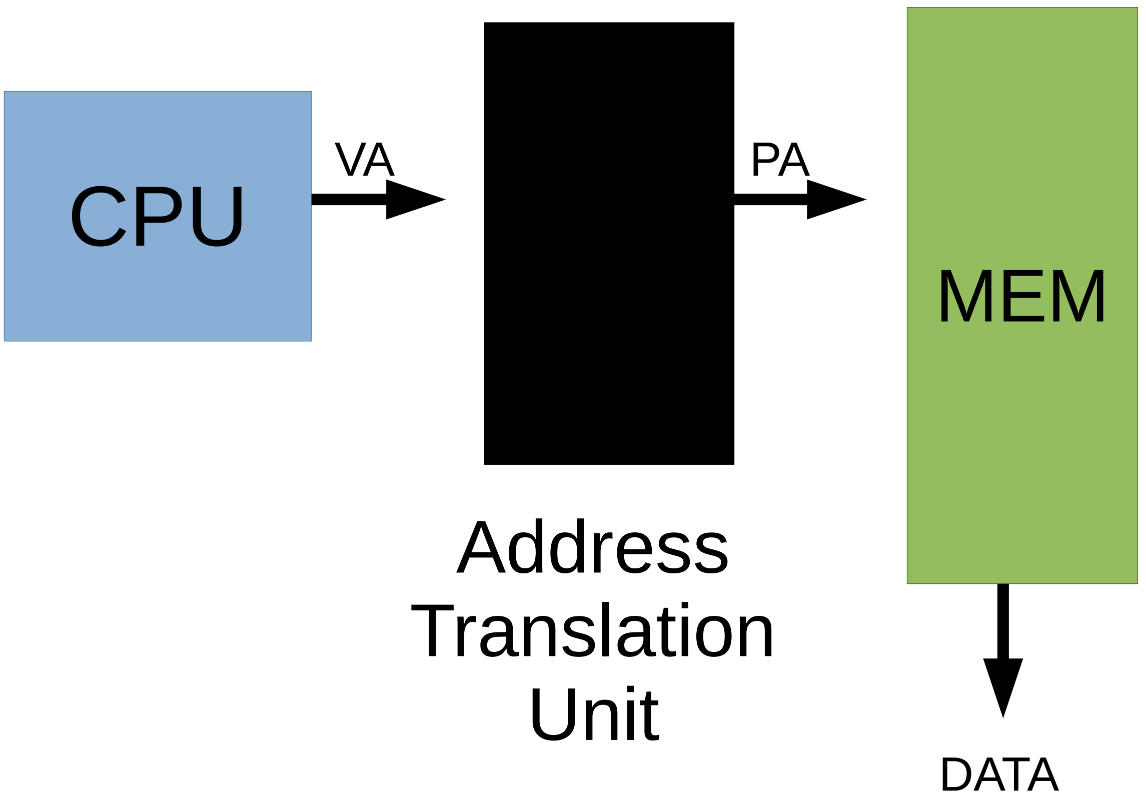
Virtual to Physical Address Translation

- Address Translation Unit determines Physical Address



Virtual to Physical Address Translation

- MMU issues memory I/O to retrieve data



Paging

- Divide each memory address into pages

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

- Example: a 16 bit address
- Assume Page Size = 512B

Paging

- Divide each memory address into pages

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

Page Number

Page Offset

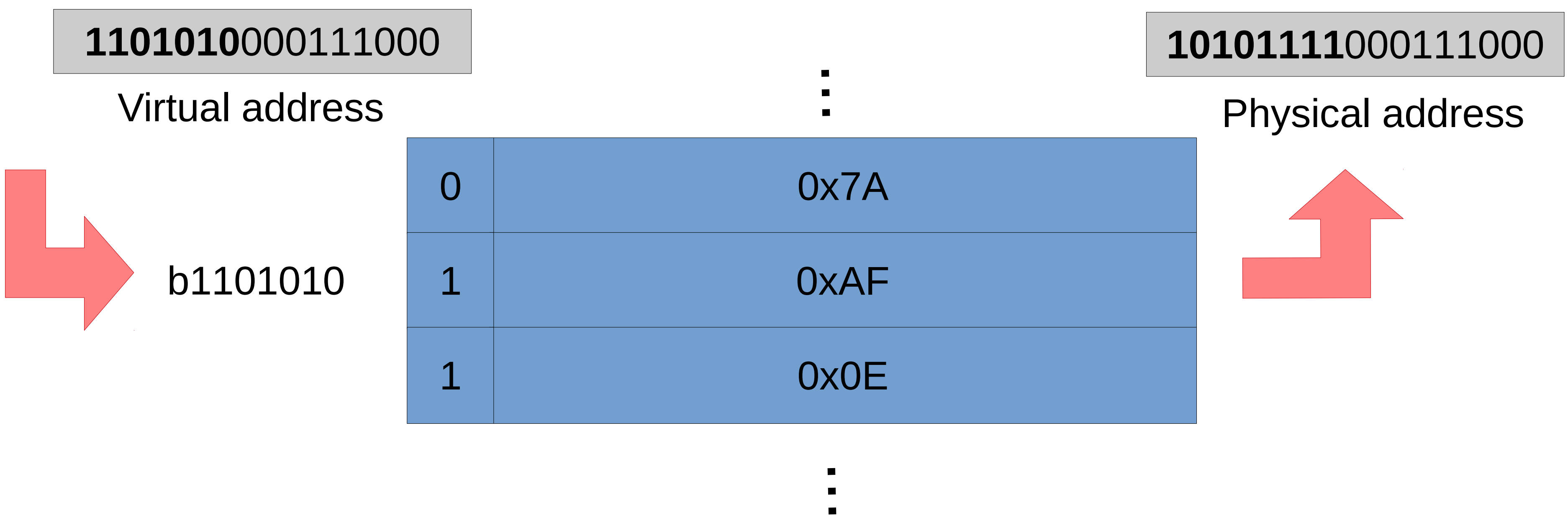
- Example: a 16 bit address
- Assume Page Size = 512B

  → 9 bit Page Offset

  → 16 – 9 = 7 bit Page Number

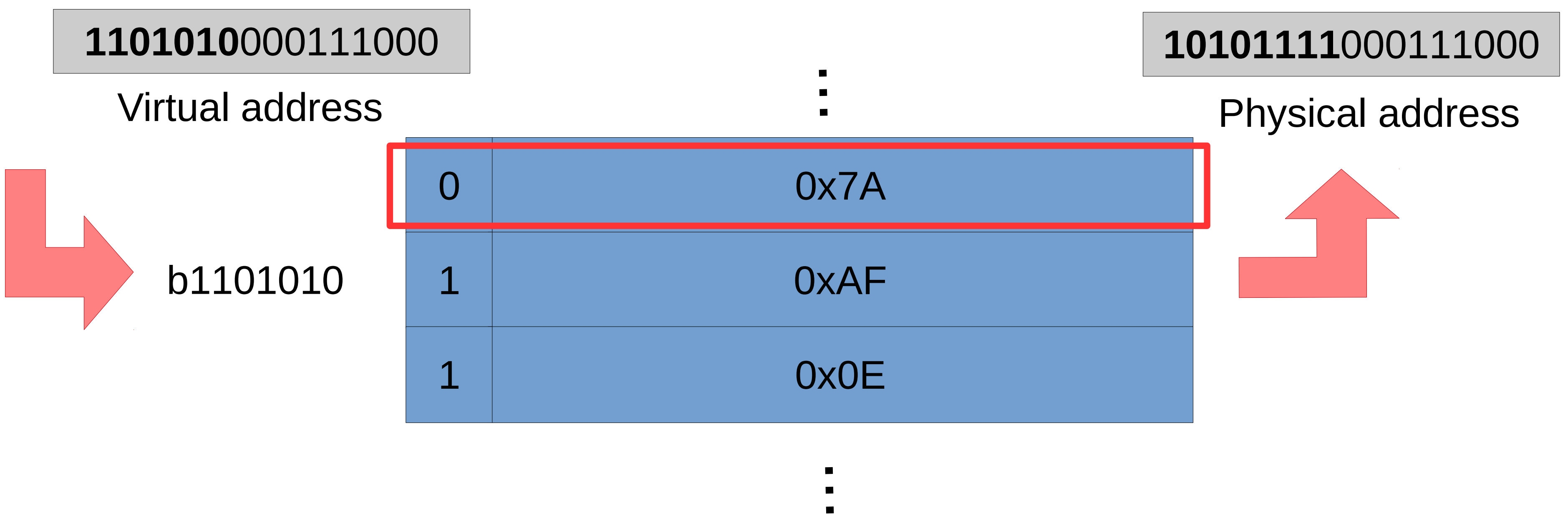
Address Translation: Page Table

- Stores 1-1 mapping from virtual page number to physical page number
  - Page offset is the same



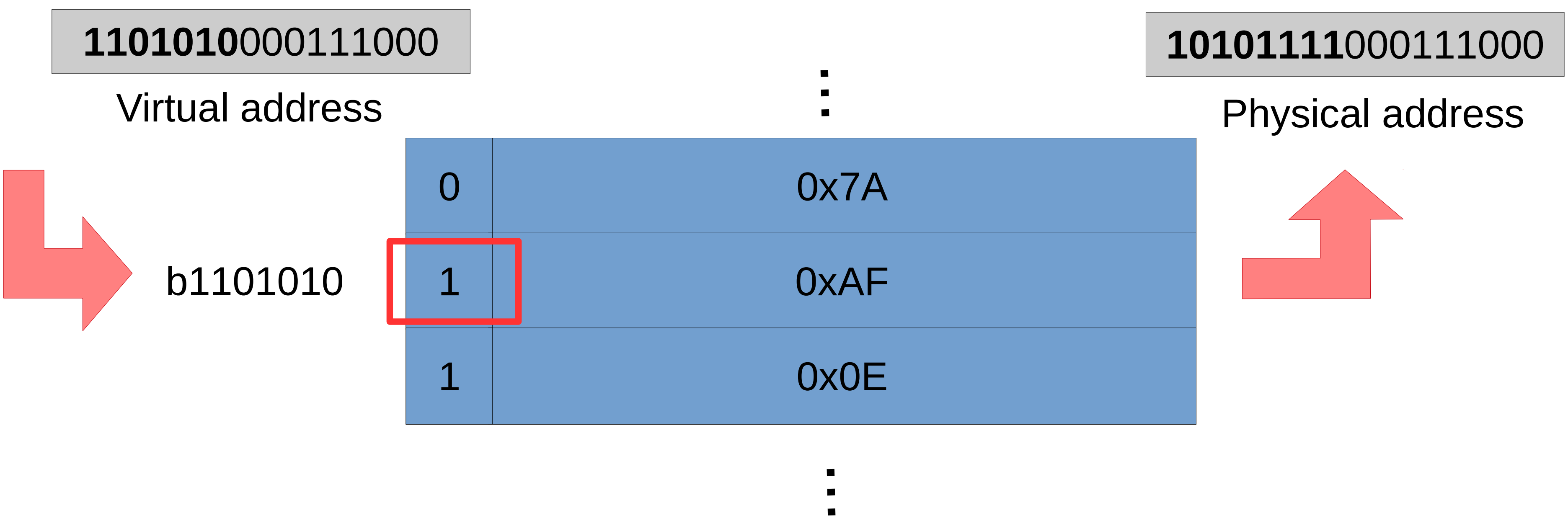
Address Translation: Page Table

- One **Page Table Entry** contains Physical Page number, and valid bit



Address Translation: Page Table

- **Valid bit** indicates whether there is a physical page that exists for the requested virtual page



Address Translation: Page Table

- **Physical Page Number** represents the real location of the page in Main Mem

I must do one PAGE TABLE lookup for every access to a virtual page

...

I must do one PAGE TABLE lookup for every access to a virtual page

…

AND a virtual page → physical page is a **one-to-one** mapping

So there is **ONE** <u>page table entry</u> **for every** virtual page

Virtual Memory: Example 1

- 16 bit Virtual Address
- 18 bit Physical Address
- Page Size P = 8 KB

  Q1: How many page table entries do we have?

Virtual Memory: Example 1

- 16 bit Virtual Address
- 18 bit Physical Address
- Page Size P= 8 KB

  Q1: How many page table entries do we have

  – Well, I need $\log_2(P) = 13$ bits for my page offset

Virtual Memory: Example 1

- 16 bit Virtual Address
- 18 bit Physical Address
- Page Size P= 8 KB

Q1: How many page table entries do we have

- Well, I need $\log_2(P) = 13$ bits for my page offset
- The remaining 16-13 = 3 bits for my page number

Virtual Memory: Example 1

- 16 bit Virtual Address
- 18 bit Physical Address
- Page Size P= 8 KB

  Q1: How many page table entries do we have
  - Well, I need $\log_2(P)$ = 13 bits for my page offset
  - The remaining 16-13 = 3 bits for my page number
  - I can uniquely identify 2^3 = 8 unique pages

Virtual Memory: Example 1

- 16 bit Virtual Address
- 18 bit Physical Address
- Page Size P= 8 KB

Q1: How many page table entries do we have

- Well, I need $\log_2(P) = 13$ bits for my page offset
- The remaining 16-13 = 3 bits for my page number
- I can uniquely identify 2^3 = 8 unique pages

→ I will have **8 Page Table entries**

Virtual Memory: Example 1

- 16 bit Virtual Address
- 18 bit Physical Address
- Page Size P= 8 KB

  Q2: How many bits to store physical page number?

Virtual Memory: Example 1

- 16 bit Virtual Address
- 18 bit Physical Address
- Page Size P= 8 KB

  Q2: How many bits to store physical page number?

  – $p = \log_2(P) = 13$ bits for page offset

  – (physical page #) $|PPN| = |PA| - p = 18 – 13 = 5$ bits

  → so my physical page number is **5 bits long**

Virtual Memory: Example 1

- Page Size P= 8 KB → p = 13 bits
- 16 bit Virtual Address → vpn = 3 bits
- 18 bit Physical Address → ppn = 5 bits
- 8 Page table entries
- Q3: What is the **Total Size** of my Page Table?

Virtual Memory: Example 1

- Page Size P= 8 KB → p = 13 bits
- 16 bit Virtual Address → vpn = 3 bits
- 18 bit Physical Address → ppn = 5 bits
- 8 Page table entries
- Q3: What is the **Total Size** of my Page Table?
  - One entry contains a **Physical Page Number** and a **valid bit**
  - So one entry is |ppn| + 1 = 5+1 = **6 bits**

Virtual Memory: Example 1

- Page Size P= 8 KB → p = 13 bits
- 16 bit Virtual Address → vpn = 3 bits
- 18 bit Physical Address → ppn = 5 bits
- 8 Page table entries
- Q3: What is the **Total Size** of my Page Table?
  - One entry contains a **Physical Page Number** and a **valid bit**
  - So one entry is |ppn| + 1 = 5+1 = **6 bits**
  - I have 8 PT entries so total size = 8x6 = **48 bits**

Virtual Memory: Example 2

- Use same PT properties from Example 1
- 8 PT entries, 16b VA, 18b PA, 4KB Page size
- Given the PT below, determine the PA of:

VA = 0xABCD

0	1	b10010
1	0	b10101
2	0	b00101
3	1	b10000
4	1	b00011
5	1	b01111
6	0	b00111
7	1	b00001

Virtual Memory: Example 2

- 8 PT entries, 16b VA, 18b PA, 4KB Page size
- Given the PT below, determine the PA of:

VA b1010101111001101

0	1	b10010
1	0	b10101
2	0	b00101
3	1	b10000
4	1	b00011
5	1	b01111
6	0	b00111
7	1	b00001

Step 1: write in binary

Virtual Memory: Example 2

- 8 PT entries, 16b VA, 18b PA, 4KB Page size
- Given the PT below, determine the PA of:

VA b**101**0101111001101

VPN = **b101**

Step 2: determine Virtual Page #

0	1	b10010
1	0	b10101
2	0	b00101
3	1	b10000
4	1	b00011
5	1	b01111
6	0	b00111
7	1	b00001

Virtual Memory: Example 2

- 8 PT entries, 16b VA, 18b PA, 4KB Page size
- Given the PT below, determine the PA of:

VA b1010101111001101

VPN = **b101 = 5**

Step 3: use VPN as index into PT to get PPN translation

0	1	b10010
1	0	b10101
2	0	b00101
3	1	b10000
4	1	b00011
5	1	b01111
6	0	b00111
7	1	b00001

Virtual Memory: Example 2

- 8 PT entries, 16b VA, 18b PA, 4KB Page size
- Given the PT below, determine the PA of:

VA b1010101111001101

PA b01111

Step 4: construct PA using PPN

0	1	b10010
1	0	b10101
2	0	b00101
3	1	b10000
4	1	b00011
5	1	b01111
6	0	b00111
7	1	b00001

Virtual Memory: Example 2

- 8 PT entries, 16b VA, 18b PA, 4KB Page size
- Given the PT below, determine the PA of:

VA b101**0101111001101**

PA b01111**0101111001101**

0	1	b10010
1	0	b10101
2	0	b00101
3	1	b10000
4	1	b00011
5	1	b01111
6	0	b00111
7	1	b00001

Step 5: **copy** page offset from VA

Virtual Memory: Example 2

- 8 PT entries, 16b VA, 18b PA, 4KB Page size
- Given the PT below, determine the PA of:

VA b101**0101111001101**

PA b01111**0101111001101**

So physical address is b011110101111001101

0	1	b10010
1	0	b10101
2	0	b00101
3	1	b10000
4	1	b00011
5	1	b01111
6	0	b00111
7	1	b00001

Virtual Memory: Example 3

- So now what about this address?

b0101111001001001

0	1	b10010
1	0	b10101
2	0	b00101
3	1	b10000
4	1	b00011
5	1	b01111
6	0	b00111
7	1	b00001

Virtual Memory: Example 3

- So now what about this address?

b0101111001001001

- No Problem!

0	1	b10010
1	0	b10101
2	0	b00101
3	1	b10000
4	1	b00011
5	1	b01111
6	0	b00111
7	1	b00001

Virtual Memory: Example 3

- So now what about this address?

b0101111001001001

- No Problem!

Virtual Page # = b010

Offset = b1111001001001

0	1	b10010
1	0	b10101
2	0	b00101
3	1	b10000
4	1	b00011
5	1	b01111
6	0	b00111
7	1	b00001

Virtual Memory: Example 3

But wait, **entry is invalid**...

What should we do?

0	1	b10010
1	0	b10101
2	$\boxed{0}$	b00101
3	1	b10000
4	1	b00011
5	1	b01111
6	0	b00111
7	1	b00001

Virtual Memory: Example 3

But wait, **entry is invalid**...

What should we do?

0	1	b10010
1	0	b10101
2	$\boxed{0}$	b00101
3	1	b10000
4	1	b00011
5	1	b01111
6	0	b00111
7	1	b00001

Option 1: 

Virtual Memory: Example 3

But wait, **entry is invalid**...

What should we do?

0	1	b10010
1	0	b10101
2	$\boxed{0}$	b00101
3	1	b10000
4	1	b00011
5	1	b01111
6	0	b00111
7	1	b00001

Actually, this is OK, we call it a **Page Fault**

So then what is a Page Fault?

- A **Page Fault** is what happens when there is **no** virtual → physical addr translation available

So then what is a Page Fault?

- A **Page Fault** is what happens when there is **no** virtual → physical addr translation available
  - e.g page we want is not in memory (yet)

So then what is a Page Fault?

- We only move pages to memory when they are needed (**Demand Paging**)
- Much like a cache miss, first access causes a **Page Fault**

So then what is a Page Fault?

- It's also possible we don't have enough physical memory to hold entire virtual address space of <u>all</u> running processes
  - So pages are removed from time to time using **LRU**

Tiny Example

- Let |PA| = |VA| = 4 bits
- Page Size = 4B

Tiny Example

- Let |PA| = |VA| = 4 bits → 16B Address space
- Page Size = 4B

→ 2 bit page number → 4 page table entries

1	b01
1	b11
1	b00
1	b10

PT

Main Mem

ab	01	3c	d7	ff	88	96	33	1e	c7	1e	0f	00	13	77	47
0				4				8				c			

Tiny Example

```
// My C program: what is value of x?
short x;
printf("%d\n", &x);        // outputs 0xe
```

1	b01
1	b11
1	b00
1	b10

PT

Main Mem

ab	01	3c	d7	ff	88	96	33	1e	c7	1e	0f	00	13	77	47
0				4				8				c			

Tiny Example

```
// My C program: what is value of x?
short x;
printf("%d\n", &x);      // outputs 0xe
```

VA = b**1110**

1	b01
1	b11
1	b00
1	b10

PT

Main Mem

0				4				8				c			
ab	01	3c	d7	ff	88	96	33	1e	c7	1e	0f	00	13	77	47

Tiny Example

```
// My C program: what is value of x?
short x;
printf("%d\n", &x);       // outputs 0xe
```

VA = b**1110**
→ VPN = b**11**

1	b01
1	b11
1	b00
1	b10

PT

Main Mem

ab	01	3c	d7	ff	88	96	33	1e	c7	1e	0f	00	13	77	47
0				4				8				c			

Tiny Example

```
// My C program: what is value of x?
short x;
printf("%d\n", &x);        // outputs 0xe
```

VA = b**1110**

→ VPN = b**11** → PT lookup

1	b01
1	b11
1	$\boxed{\text{b00}}$
1	b10

PT

Main Mem

ab	01	3c	d7	ff	88	96	33	1e	c7	1e	0f	00	13	77	47
0				4				8				c			

Tiny Example

```
// My C program: what is value of x?
short x;
printf(“%d\n”, &x);      // outputs 0xe
```

VA = b**1110**

→ VPN = b**11**

→ PPN = b**00**

1	b01
1	b11
1	b00
1	b10

PT

Main Mem

ab	01	3c	d7	ff	88	96	33	1e	c7	1e	0f	00	13	77	47
0				4				8				c			

Tiny Example

```c
// My C program: what is value of x?
short x;
printf("%d\n", &x);        // outputs 0xe
```

So PA = **00****10**

PPN   Page Offset

1	b01
1	b11
1	b00
1	b10

PT

Main Mem

ab	01	3c	d7	ff	88	96	33	1e	c7	1e	0f	00	13	77	47
0				4				8				c			

Tiny Example

```
// My C program: what is value of x?
short x;
printf("%d\n", &x);        // outputs 0xe
```

1	b01
1	b11
1	b00
1	b10

PT

So x = 0xd73c

PA

Main Mem

ab	01	3c	d7	ff	88	96	33	1e	c7	1e	0f	00	13	77	47
0				4				8				c			

So does Virtual memory solve all our problems?

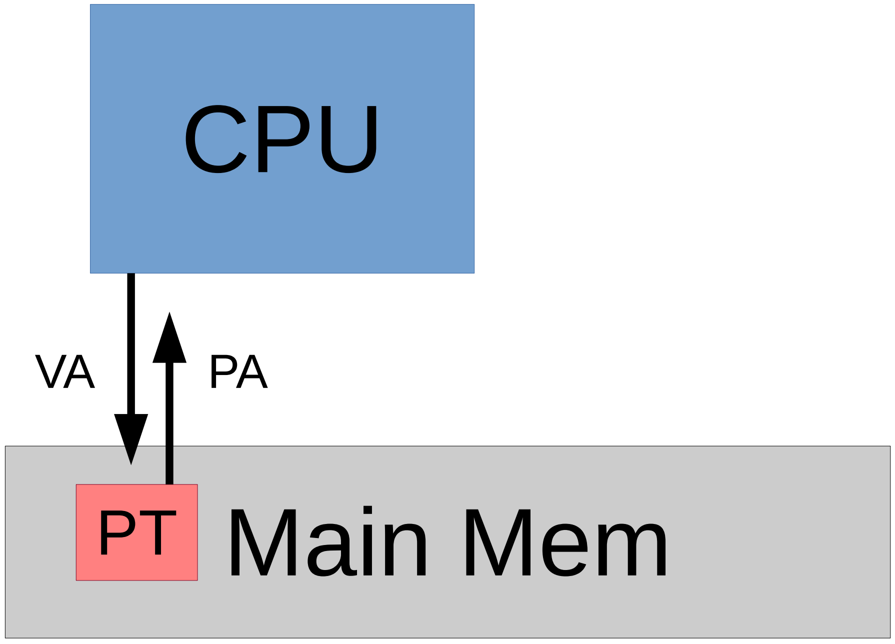
Well...no

Every memory access must first lookup translation in PT

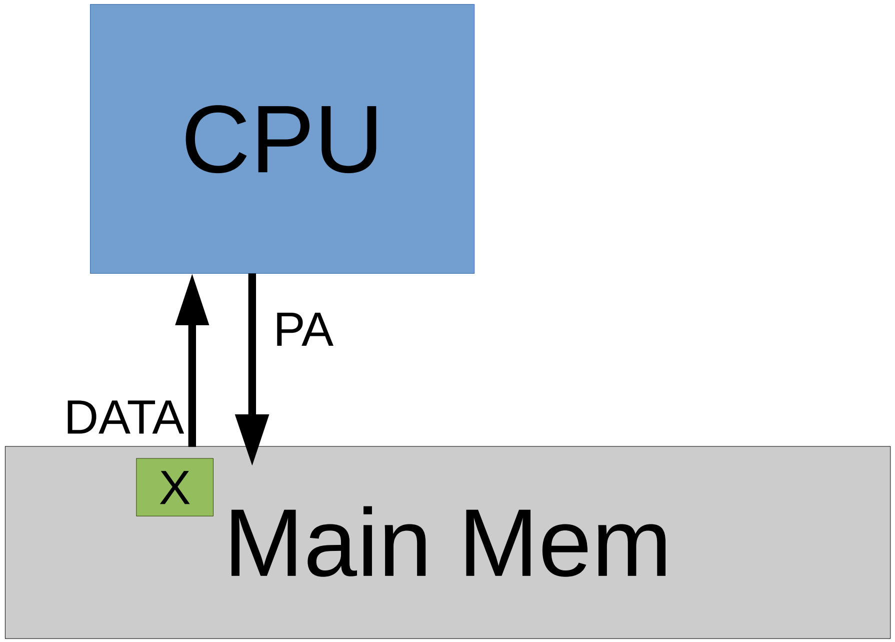
The Page Table has to live somewhere – we'll have to store it in memory

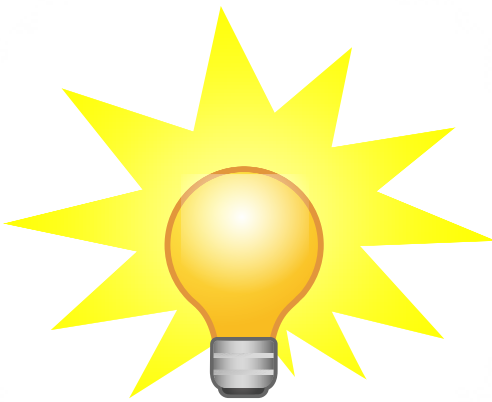
So now I have to do 2 I/O's for every memory access

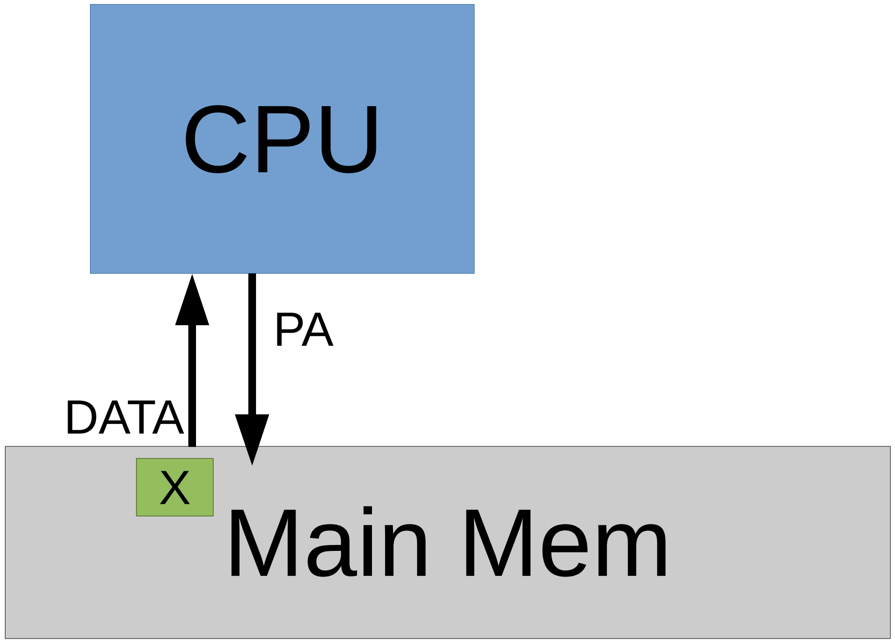
One to fetch Physical addr translation from Page Table



And another to fetch the actual data







Two Memory I/O's **per access** is very expensive...

Why don't we cache the Page Table?

Translation Lookaside Buffer

- TLB
- A fancy name for Cache
- Caches subset of page table entries
- Single cycle lookup of virtual → physical addr
- If interested in learning more, take CS M151B

Dynamic Memory Allocators (Chapter 9.9)

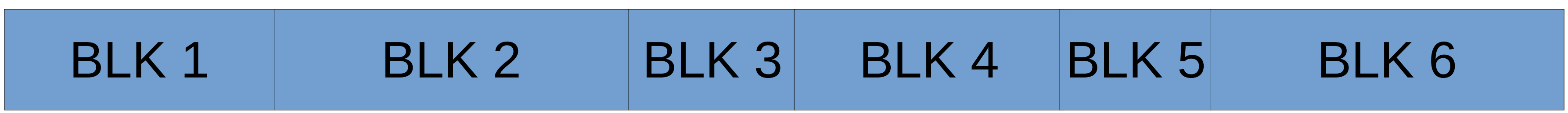
Malloc and Free

```
/* Allocate 'amt' Bytes on heap and
 * give me a ptr to access this memory
 */
void* ptr = malloc(amt);
/* Free the block starting at ptr */
free(ptr);
```

- Q: How does this work?

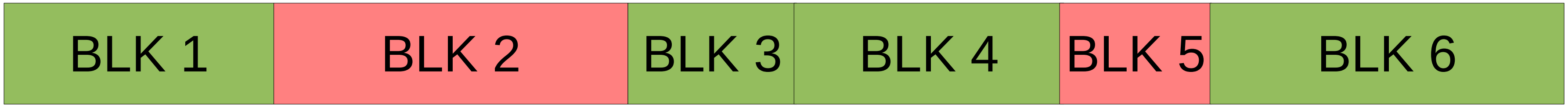
Building a Heap with Explicit Free List

- Heap is a segment of memory partitioned into blocks



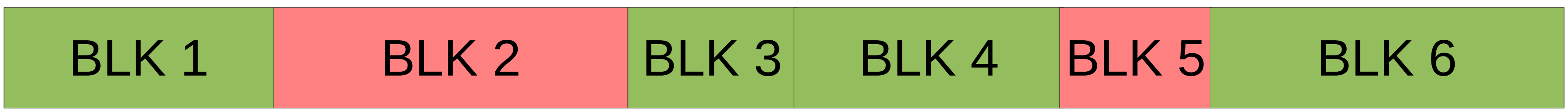
Building a Heap with Explicit Free List

- Some blocks are allocated and some are free



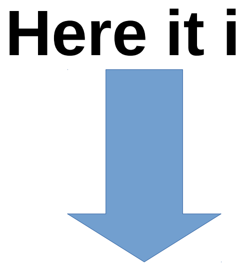
Building a Heap with Explicit Free List

```
int* ptr = malloc(1024);
```

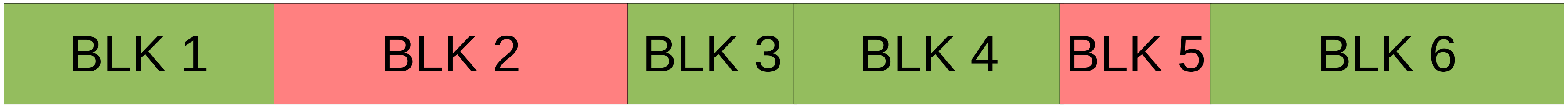


- When user issues allocation request, we have to find a free block that can fit the request

Building a Heap with Explicit Free List

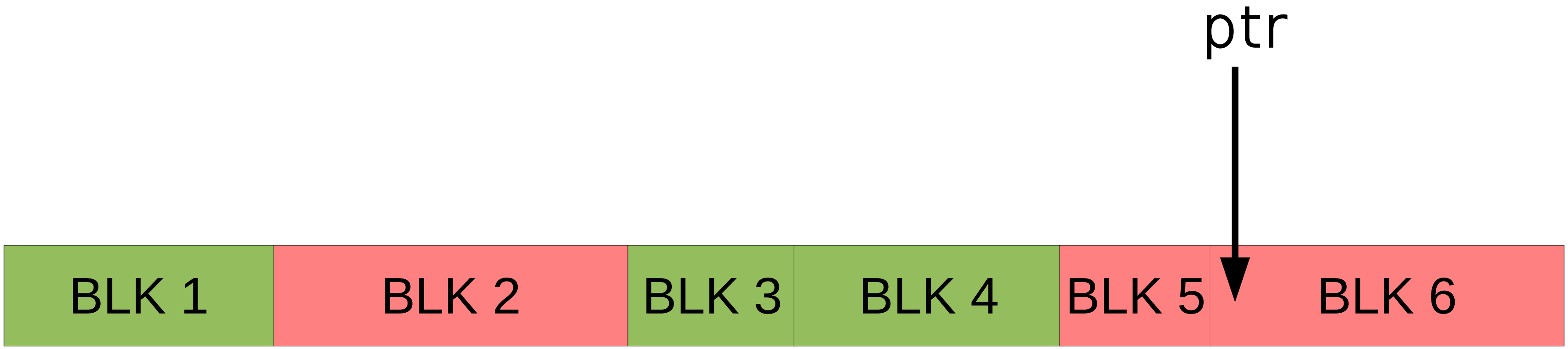



```
int* ptr = malloc(1024);
```



- Any one that is big enough will do

Building a Heap with Explicit Free List

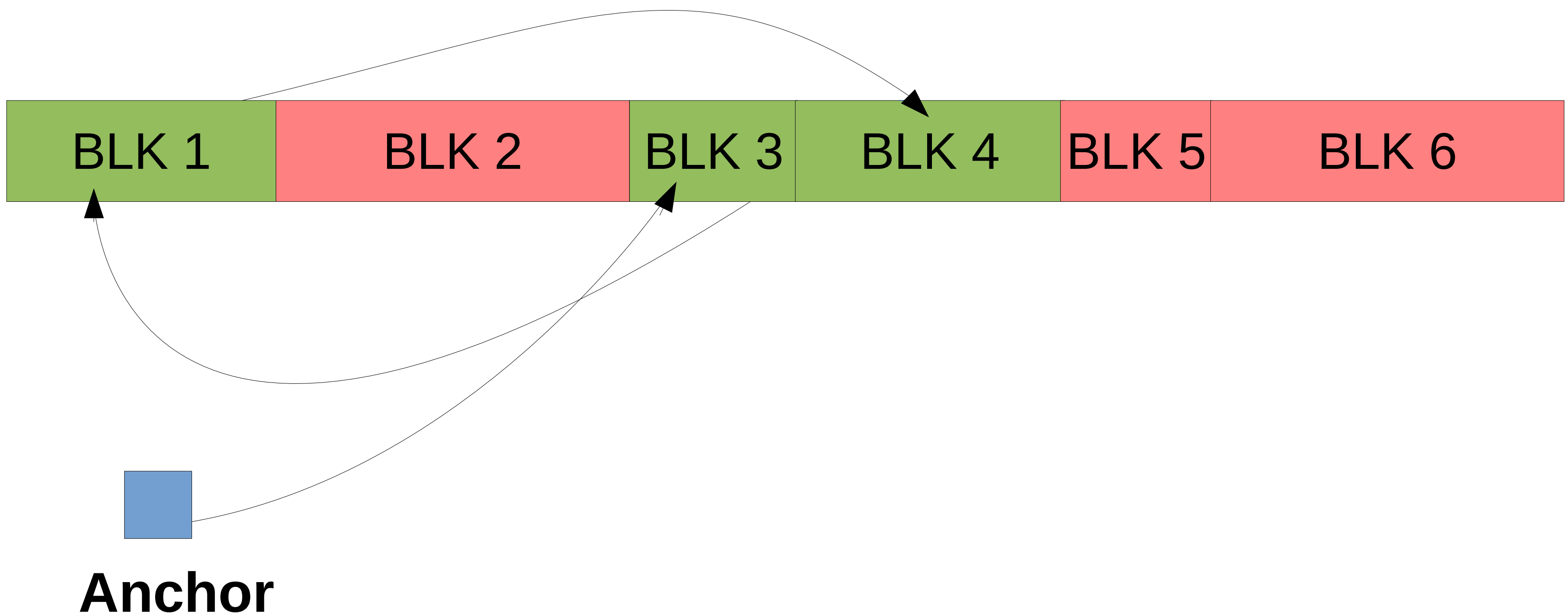


- Now we can mark it as allocated

To make searching of available free block faster, we maintain a **free list**

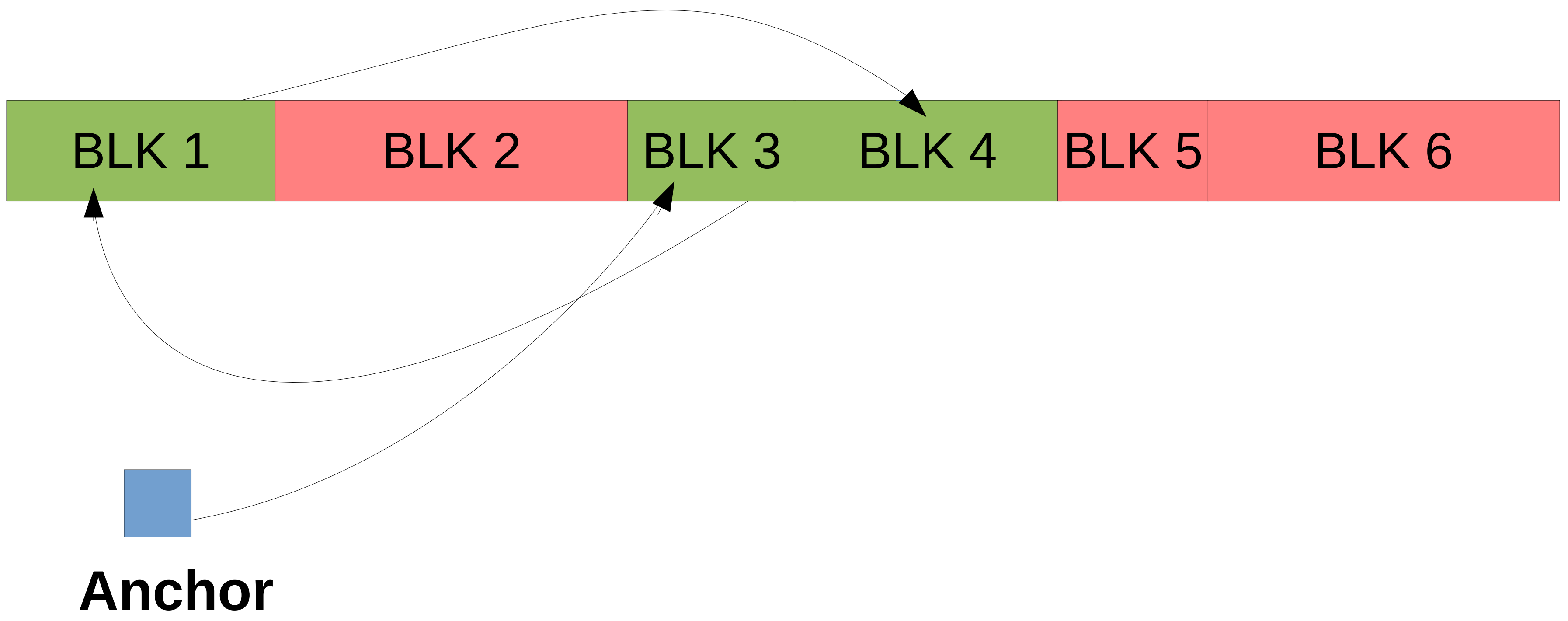
Building a Heap with Explicit Free List

- A Free List is a linked list that chains together the list of free blocks



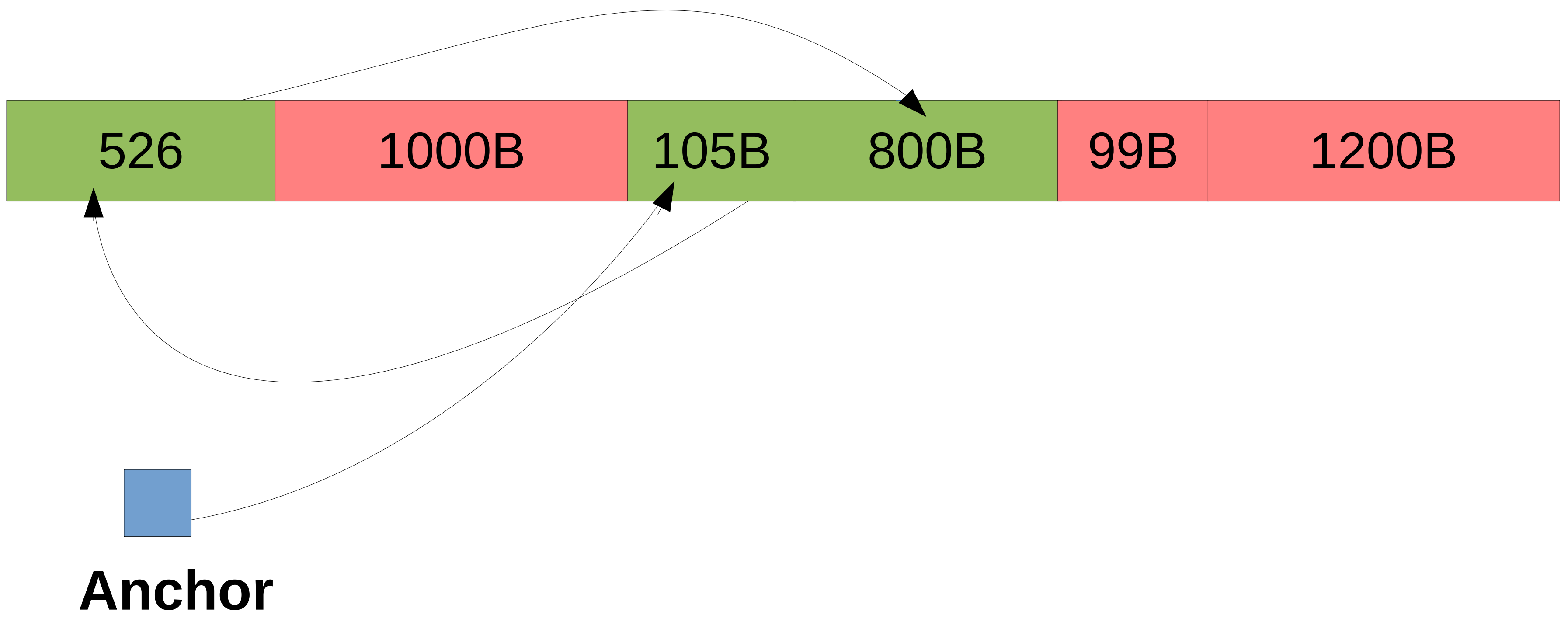
Building a Heap with Explicit Free List

- A Free List is a linked list that chains together the list of free blocks



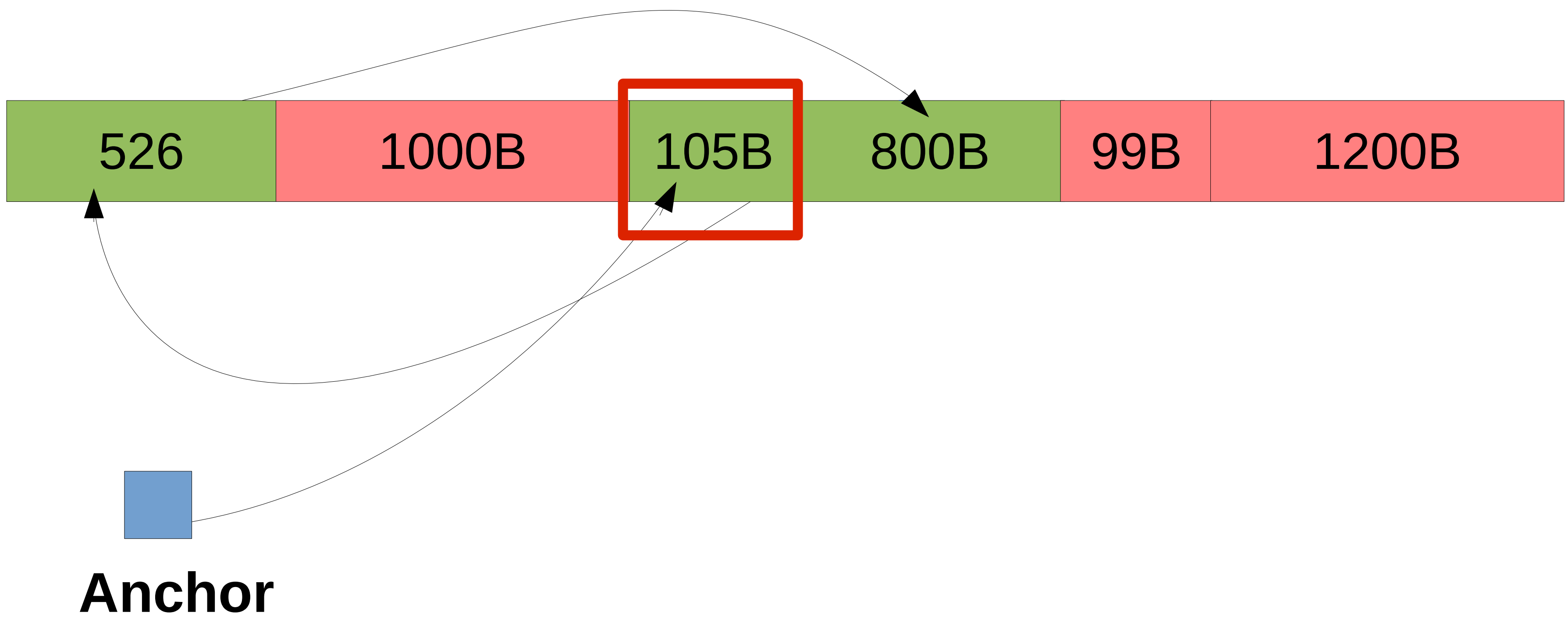
Building a Heap with Explicit Free List

- Now I receive the following allocation request: malloc(512);



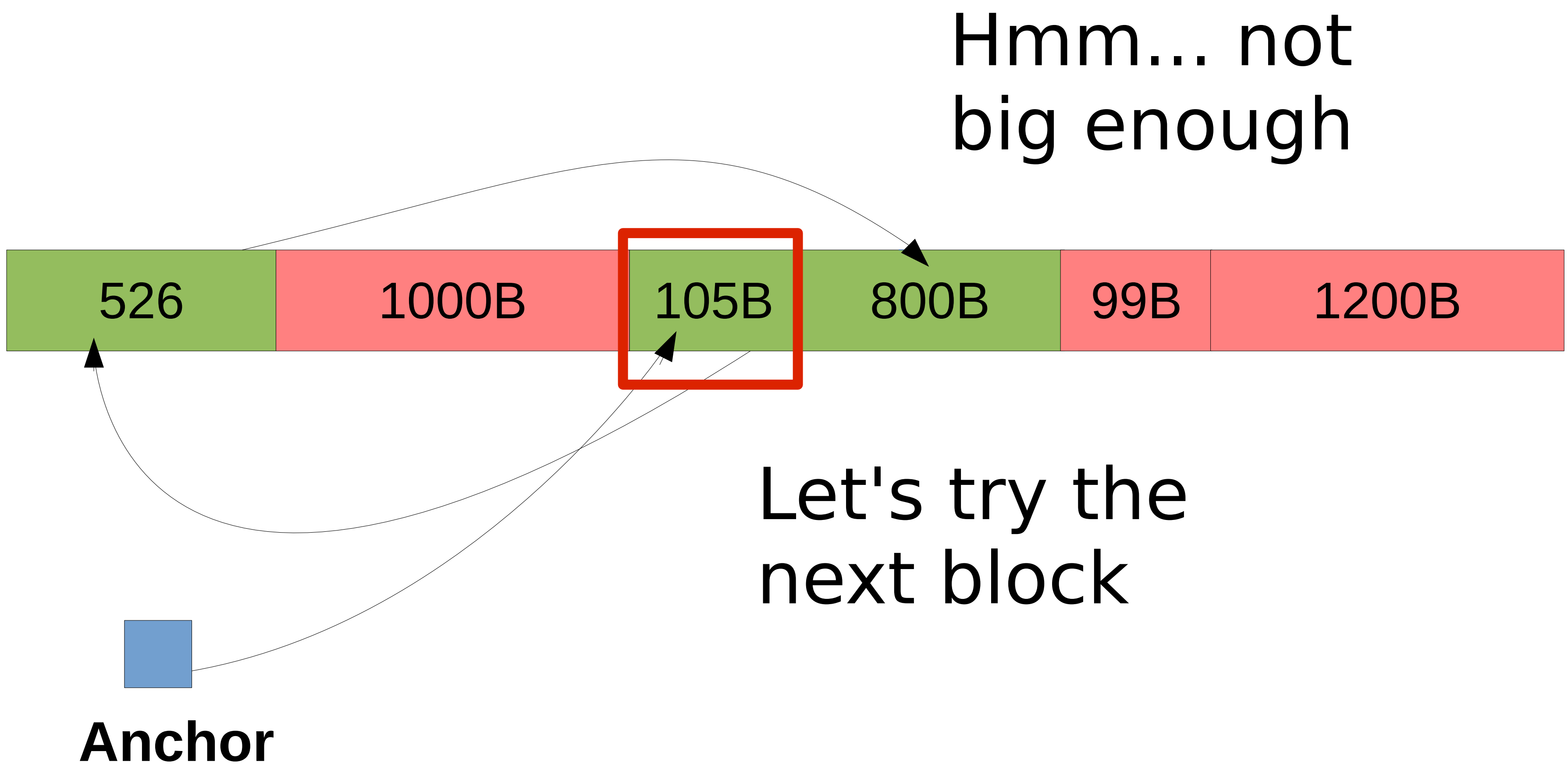
Building a Heap with Explicit Free List

- Starting at anchor, I lookup first free block



- malloc(512);

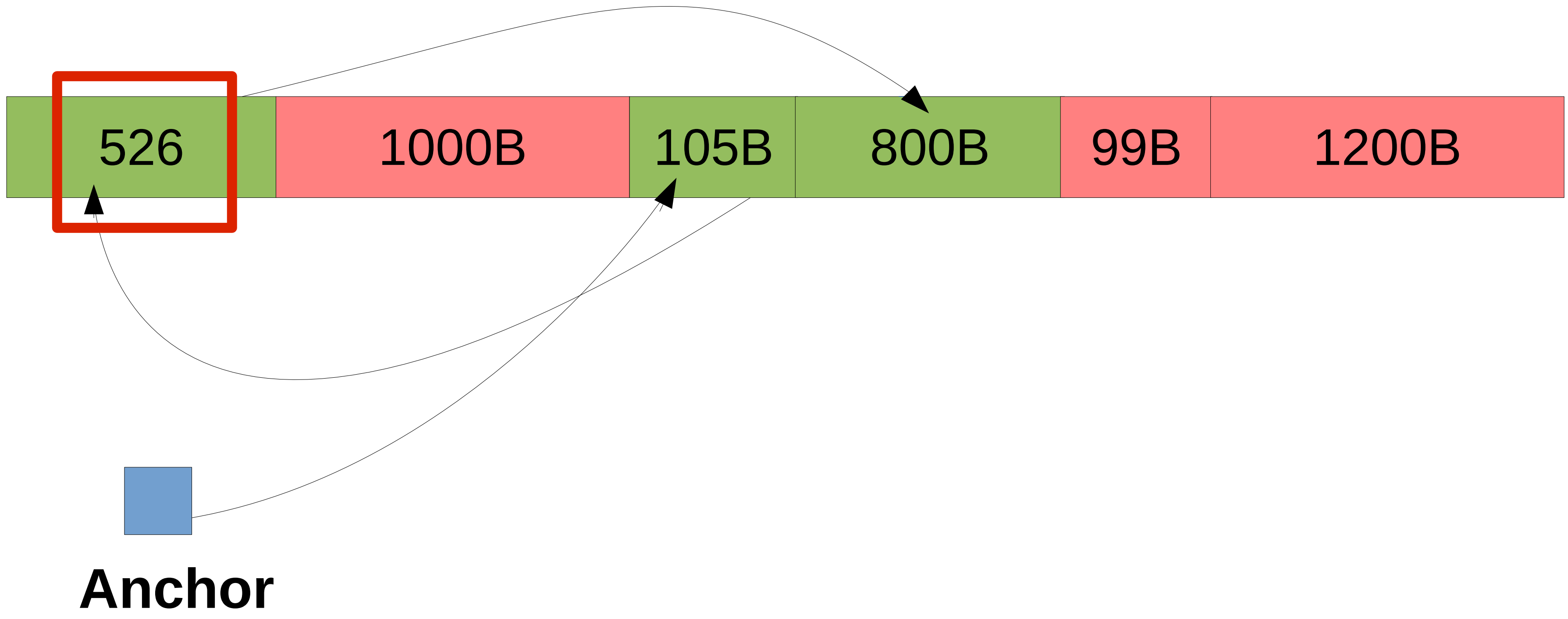
Building a Heap with Explicit Free List



- malloc(512);

Building a Heap with Explicit Free List

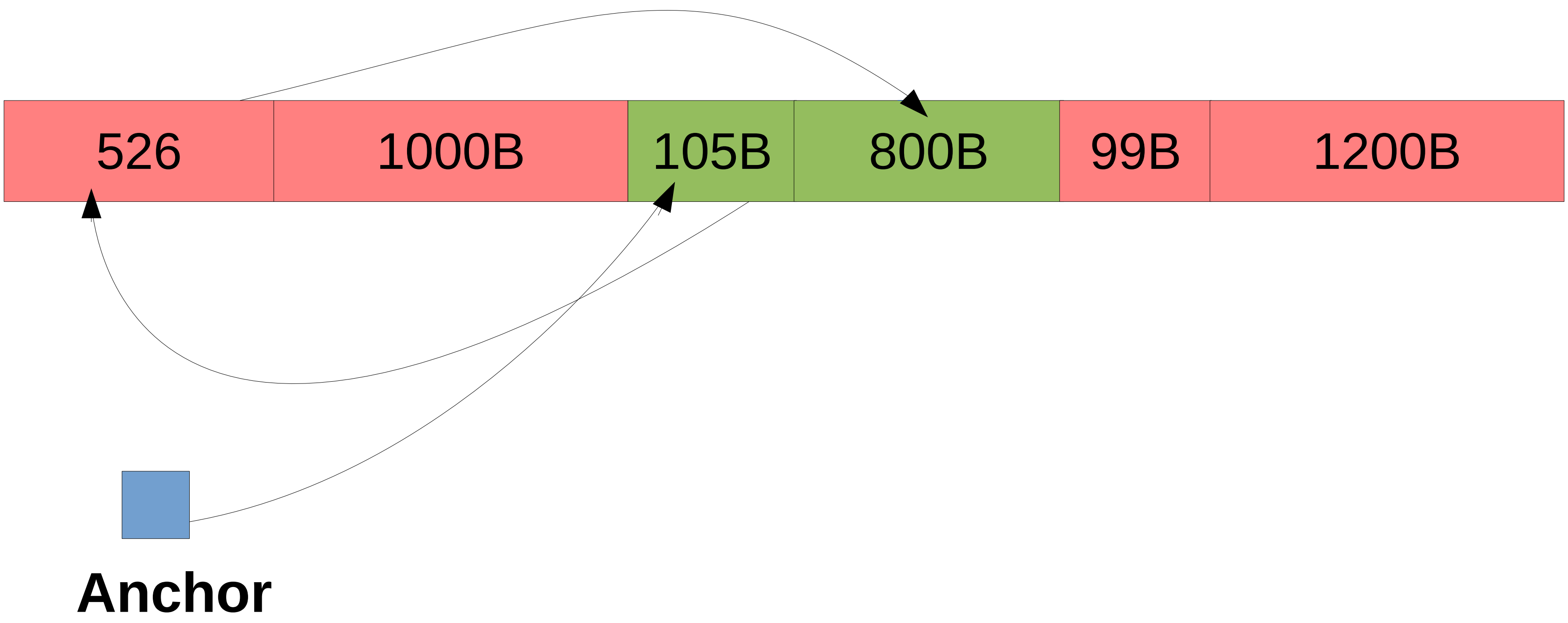
Looks good. Let's use this one.



- malloc(512);

Building a Heap with Explicit Free List

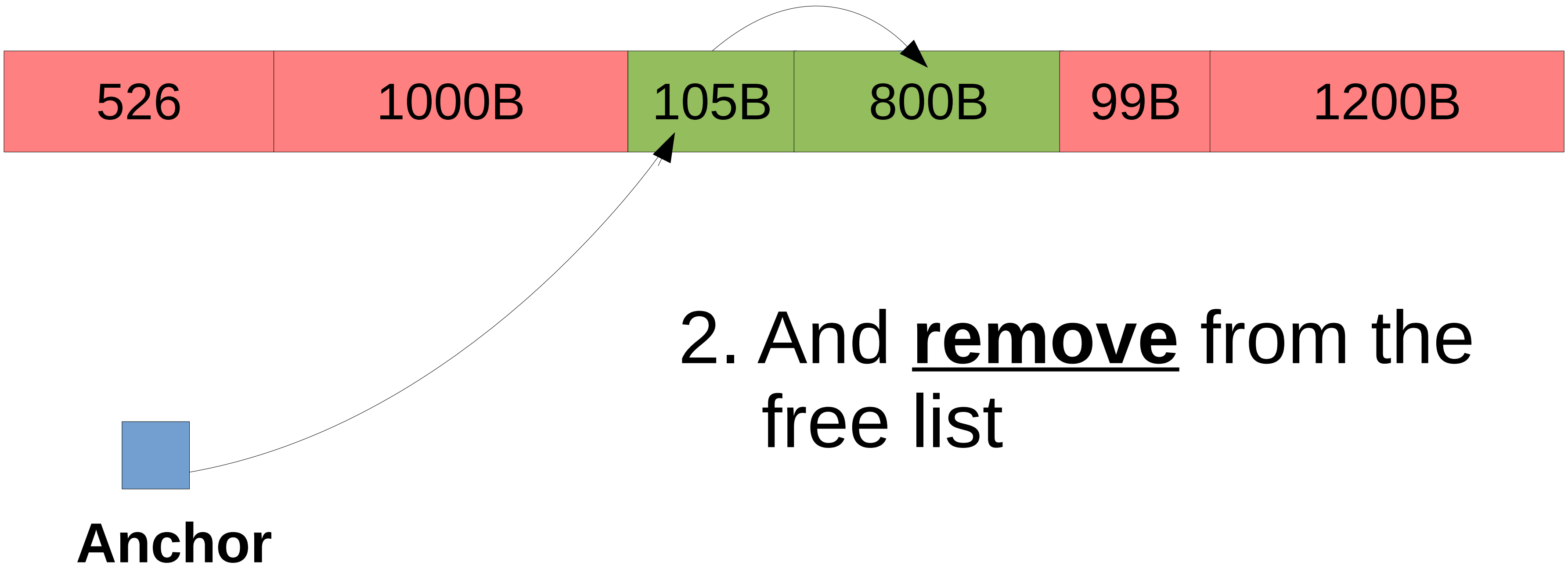
1. Mark the block as allocated



- malloc(512);

Building a Heap with Explicit Free List

1. Mark the block as allocated



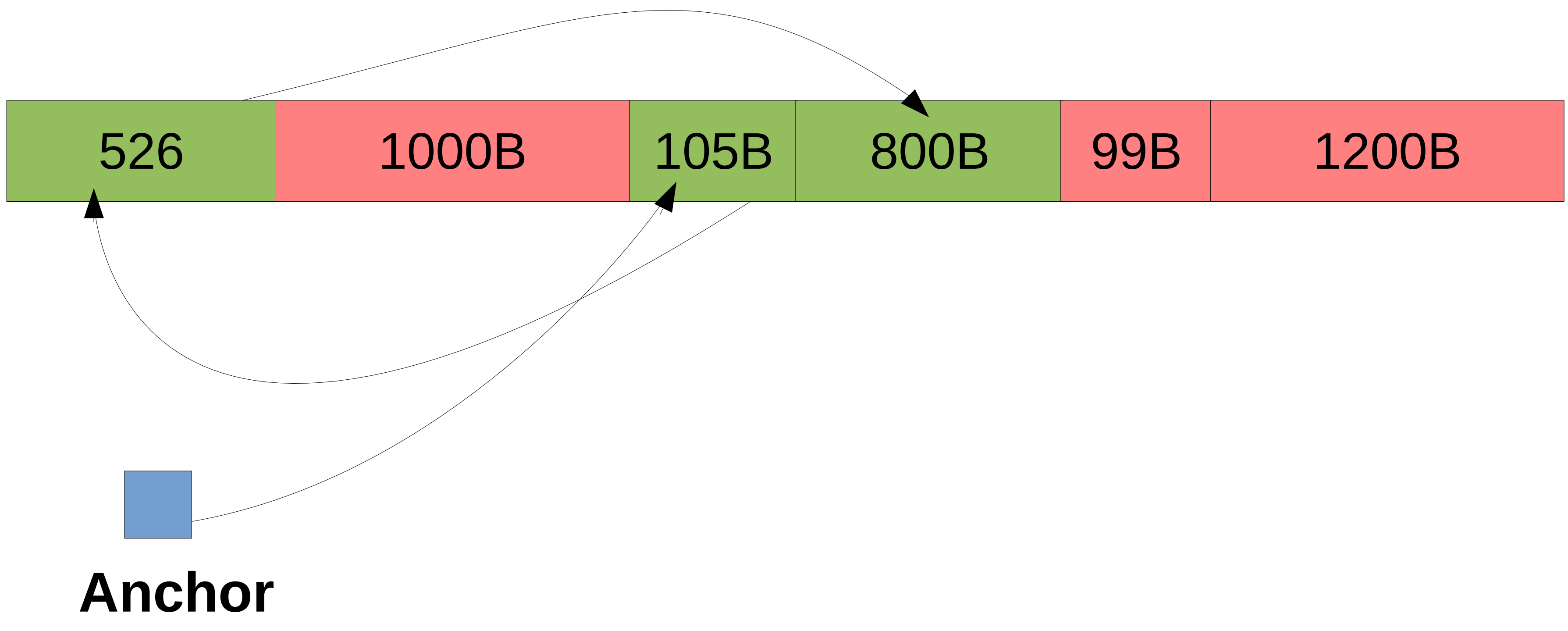
2. And **remove** from the free list

- malloc(512);

Somtimes, a free block has <u>too much</u> free space

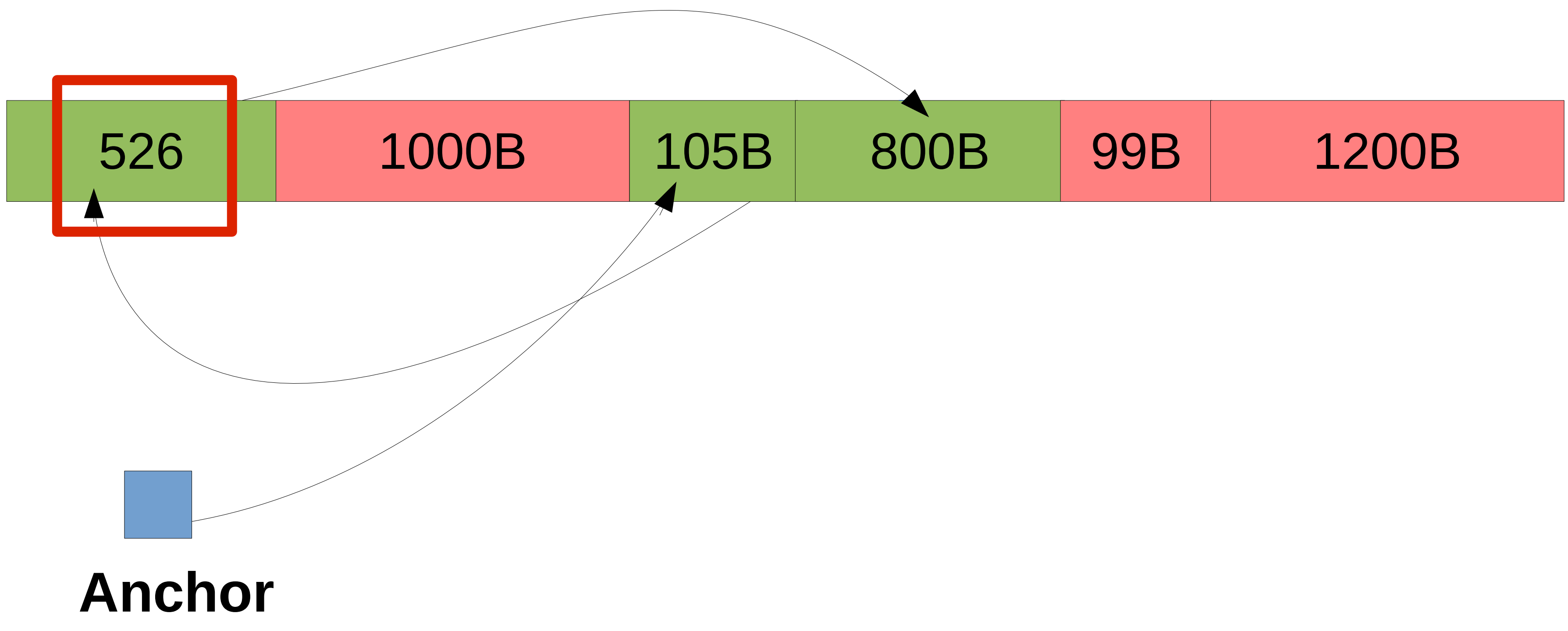
Building a Heap with Explicit Free List

- Lets say instead I say **malloc(106)**



Building a Heap with Explicit Free List

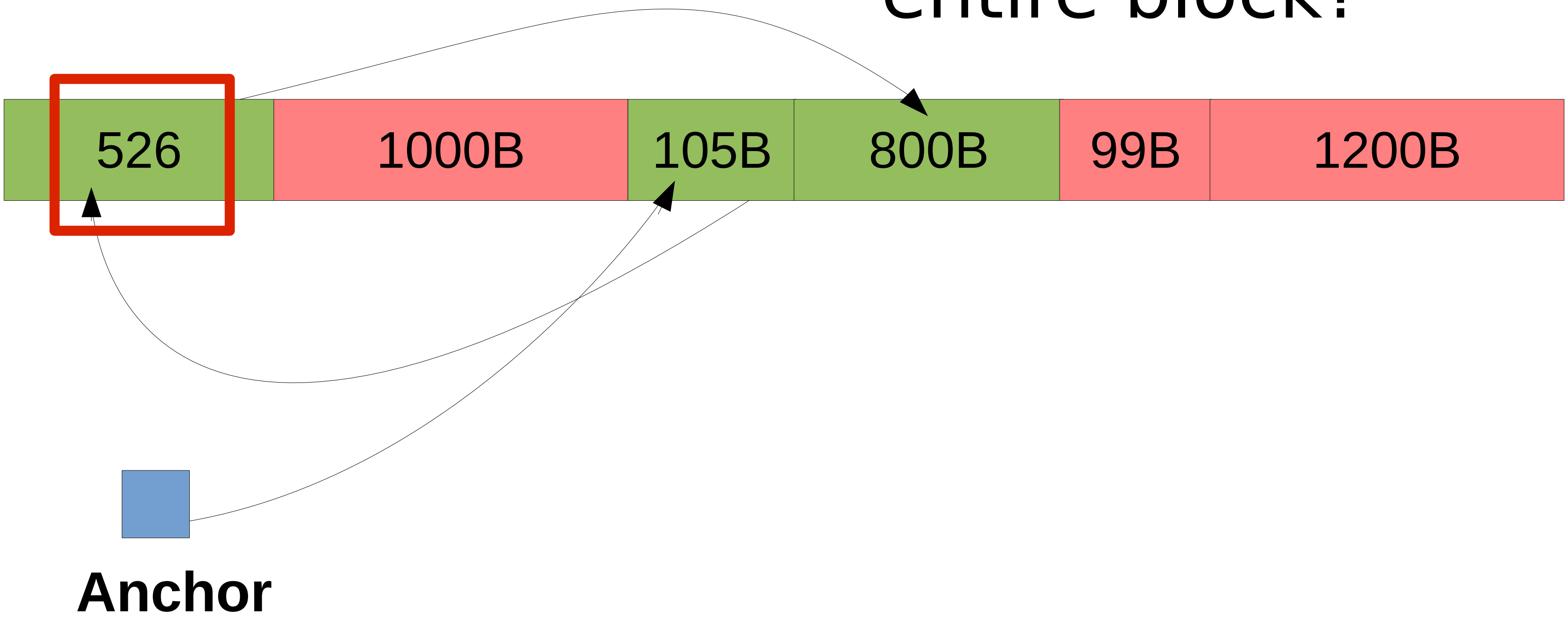
- Again, this is the candidate free block



- malloc(106);

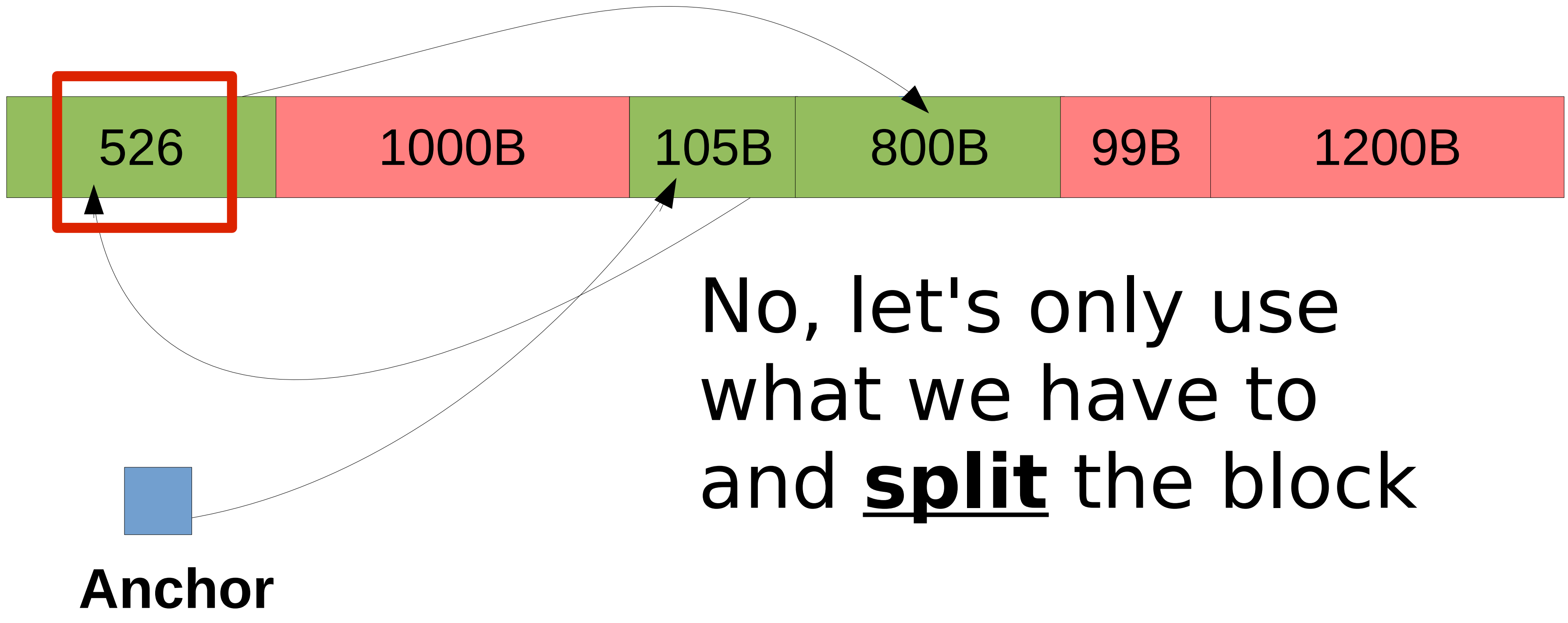
Building a Heap with Explicit Free List

But do I alloc the entire block?



- malloc(106);

Building a Heap with Explicit Free List

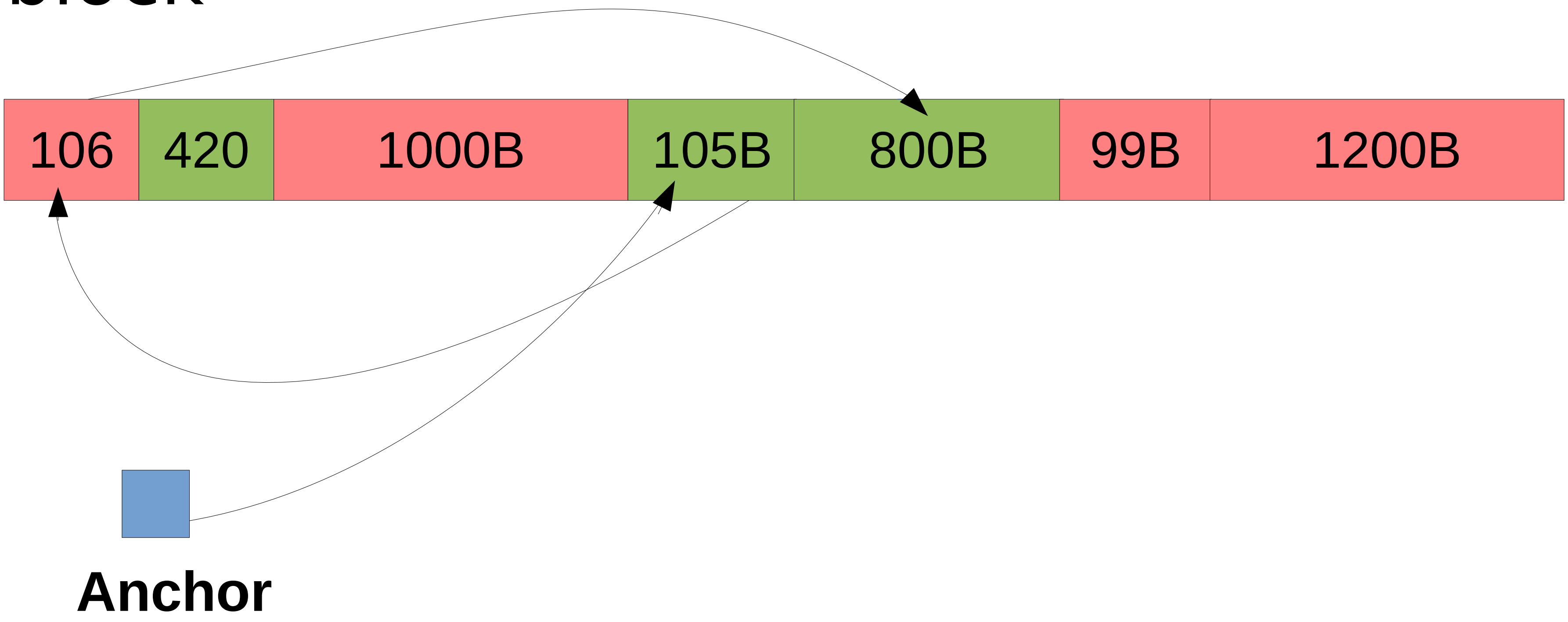


No, let's only use what we have to and **split** the block

- malloc(106);

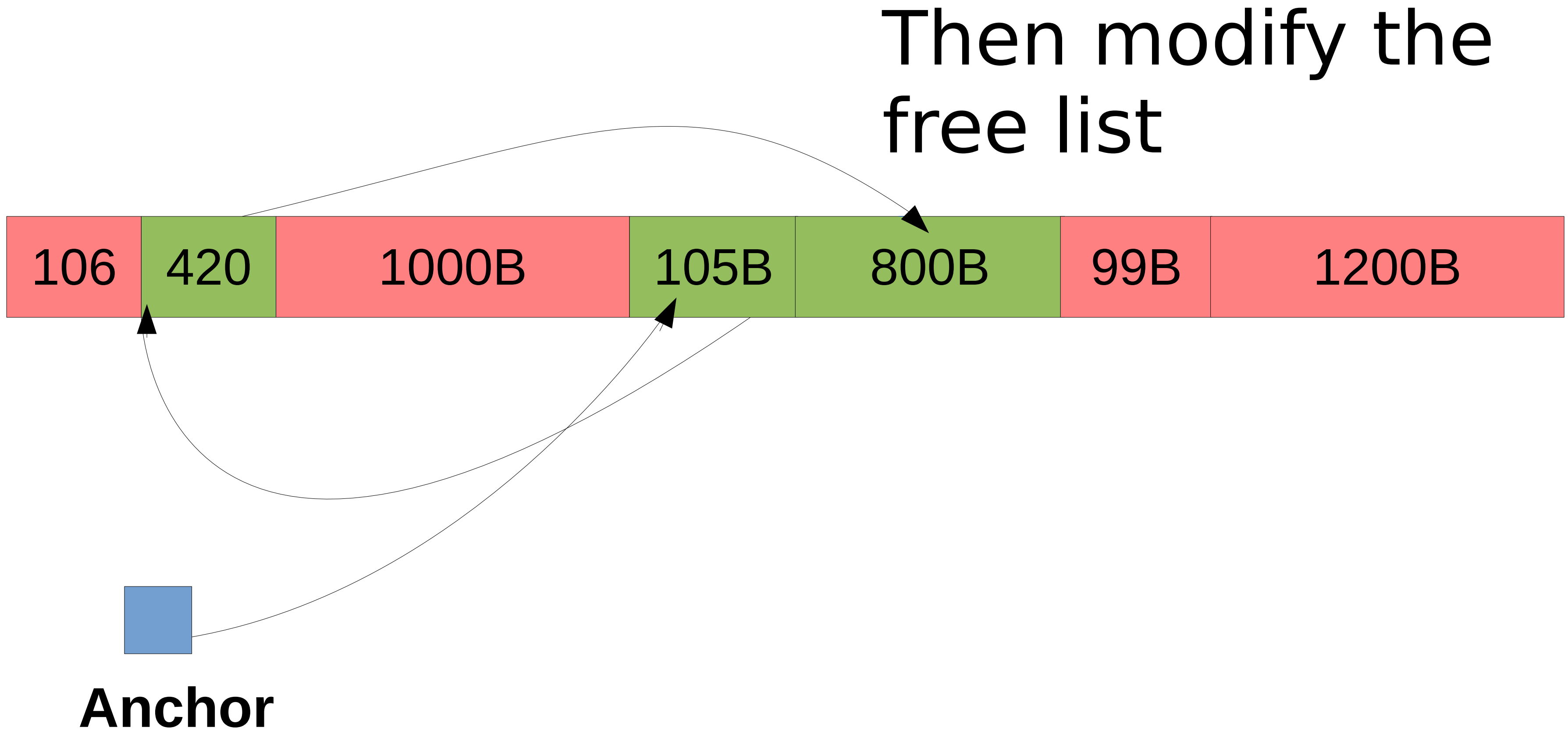
Building a Heap with Explicit Free List

Allocate the first block



- malloc(106);

Building a Heap with Explicit Free List



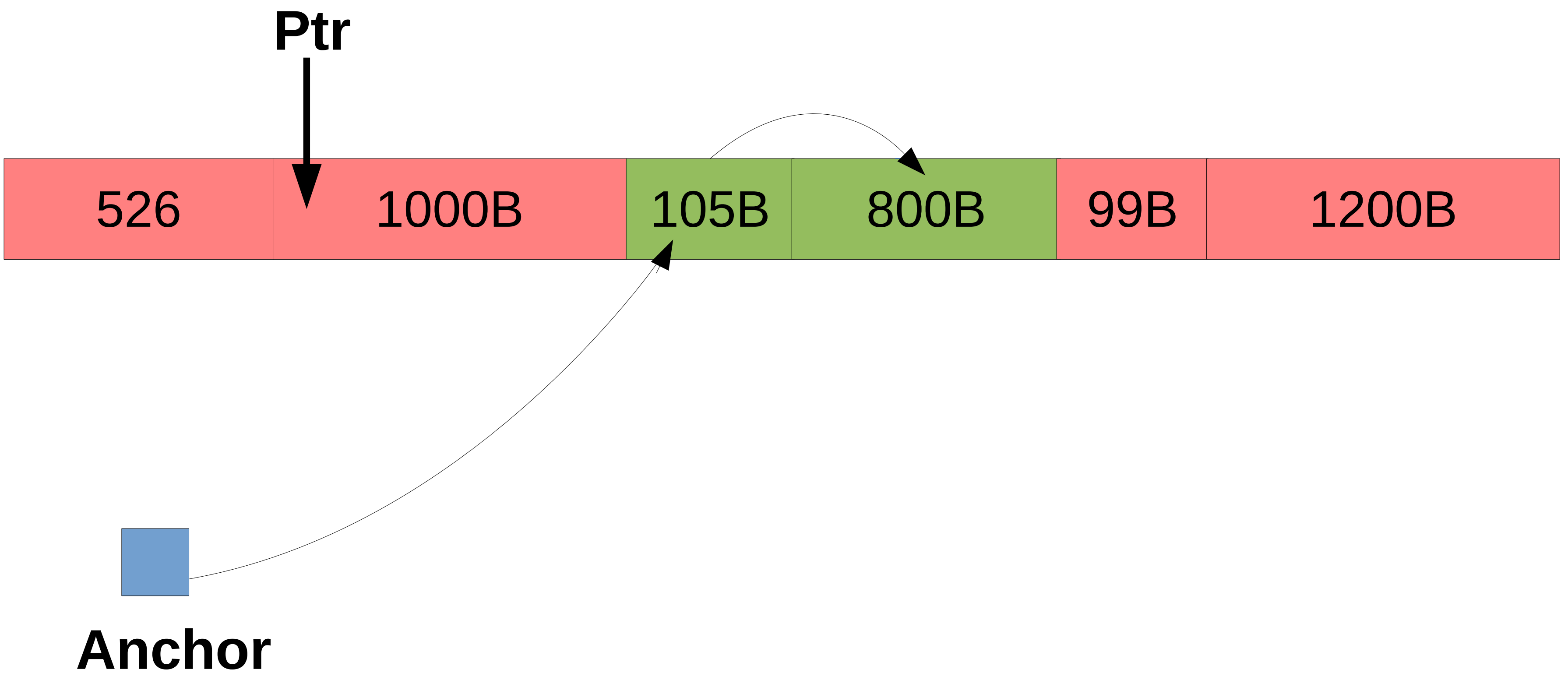
- malloc(106);

C is Lazy

We don't care about allocated blocks because it is user responsibility to free it!

Building a Heap with Explicit Free List

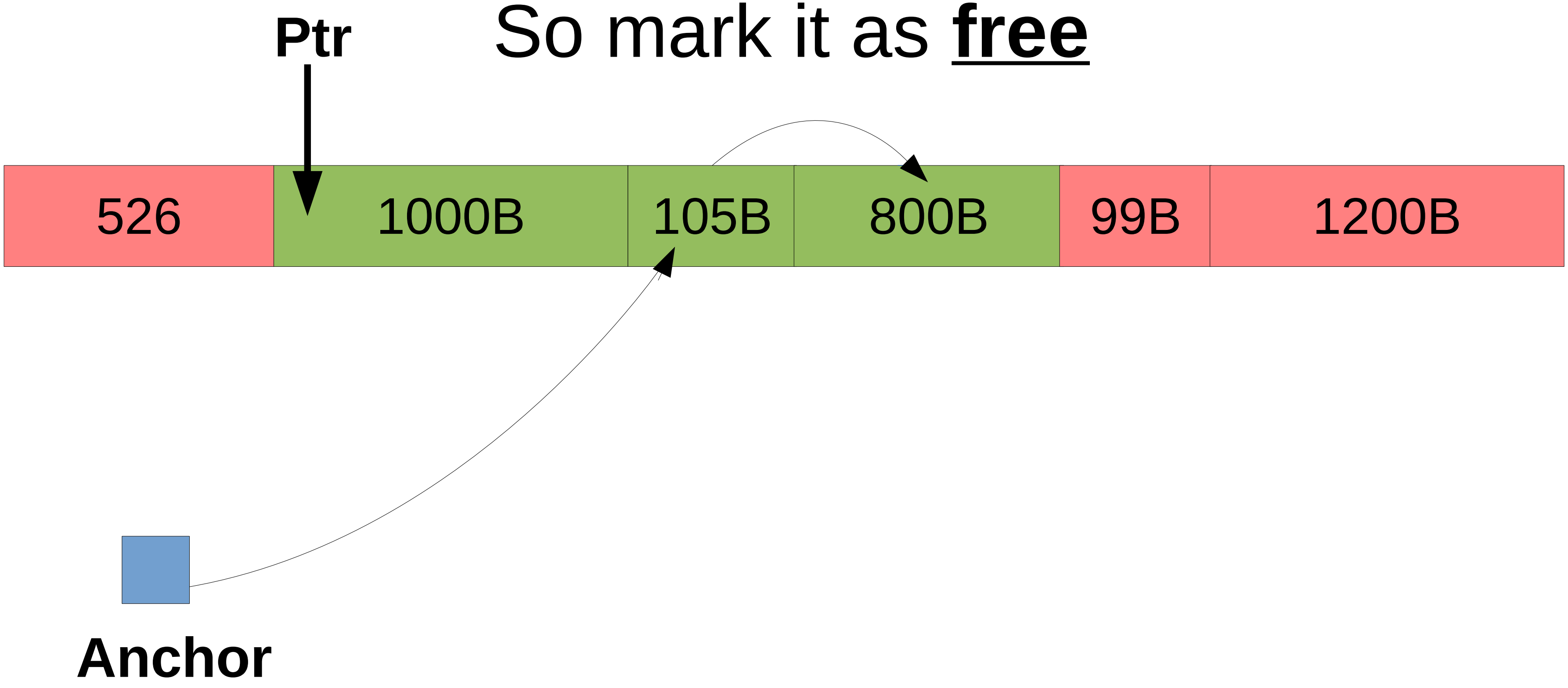
Now, user wants to free this block



free(ptr)

Building a Heap with Explicit Free List

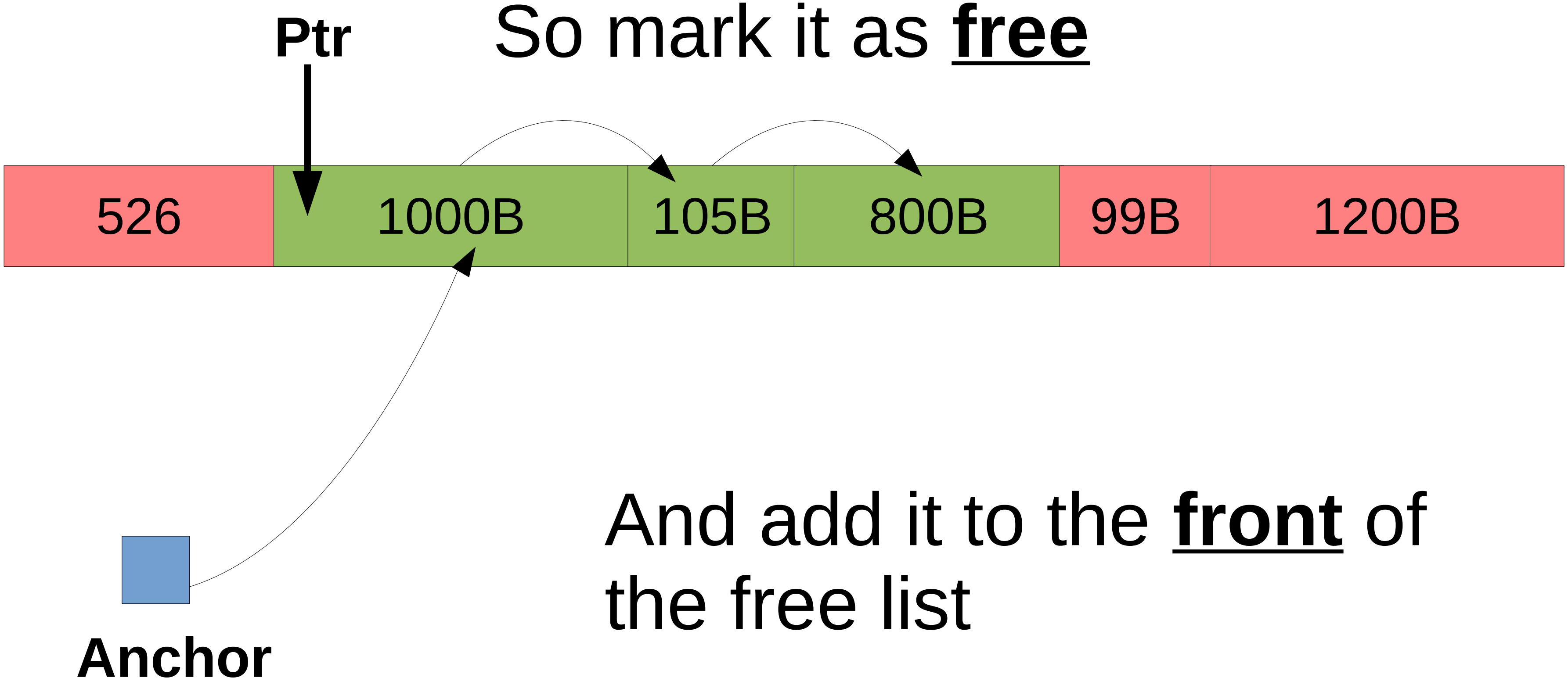
So mark it as **free**



free(ptr)

Building a Heap with Explicit Free List

So mark it as **free**

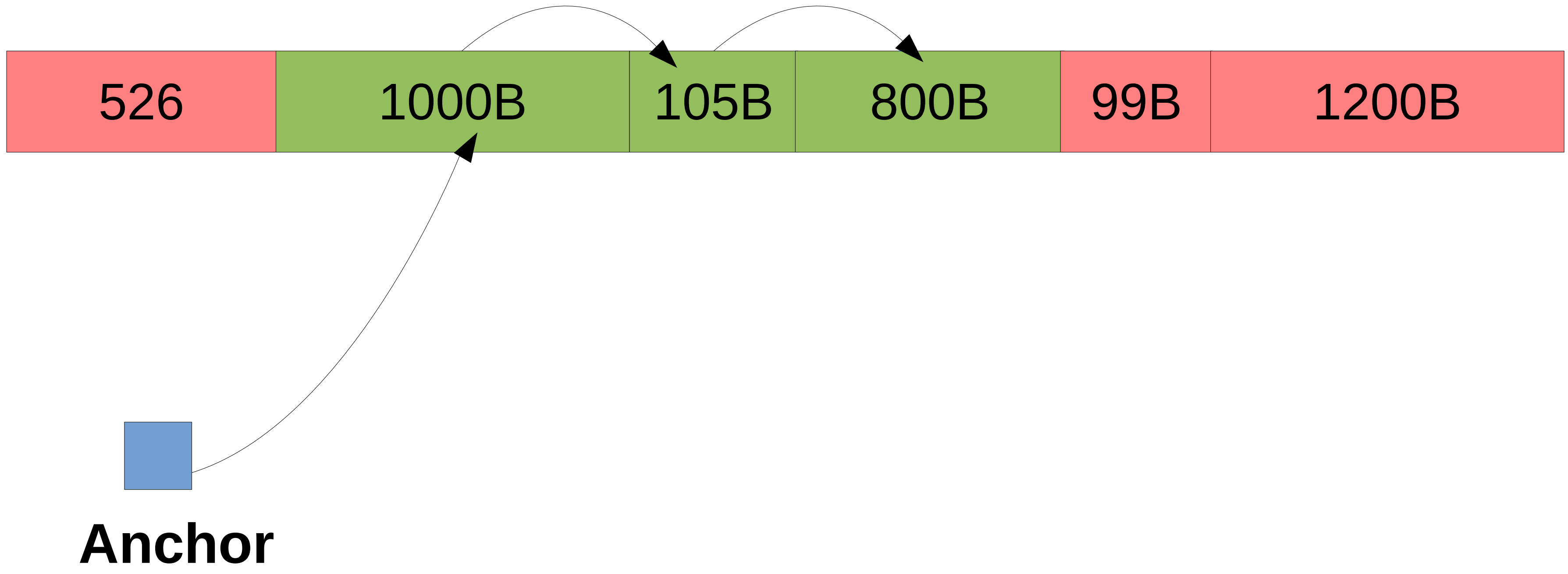


And add it to the **front** of the free list

```
free(ptr)
```

Building a Heap with Explicit Free List

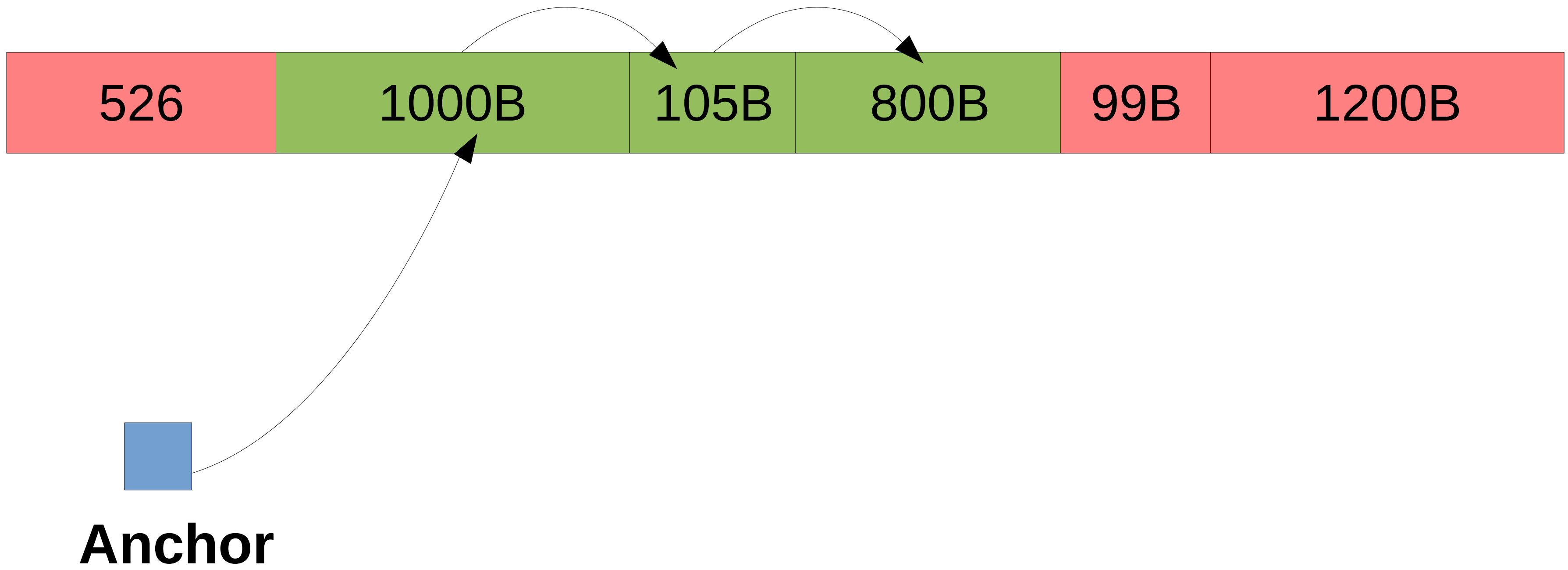
Now, user wants to malloc 1200B



malloc(1200)

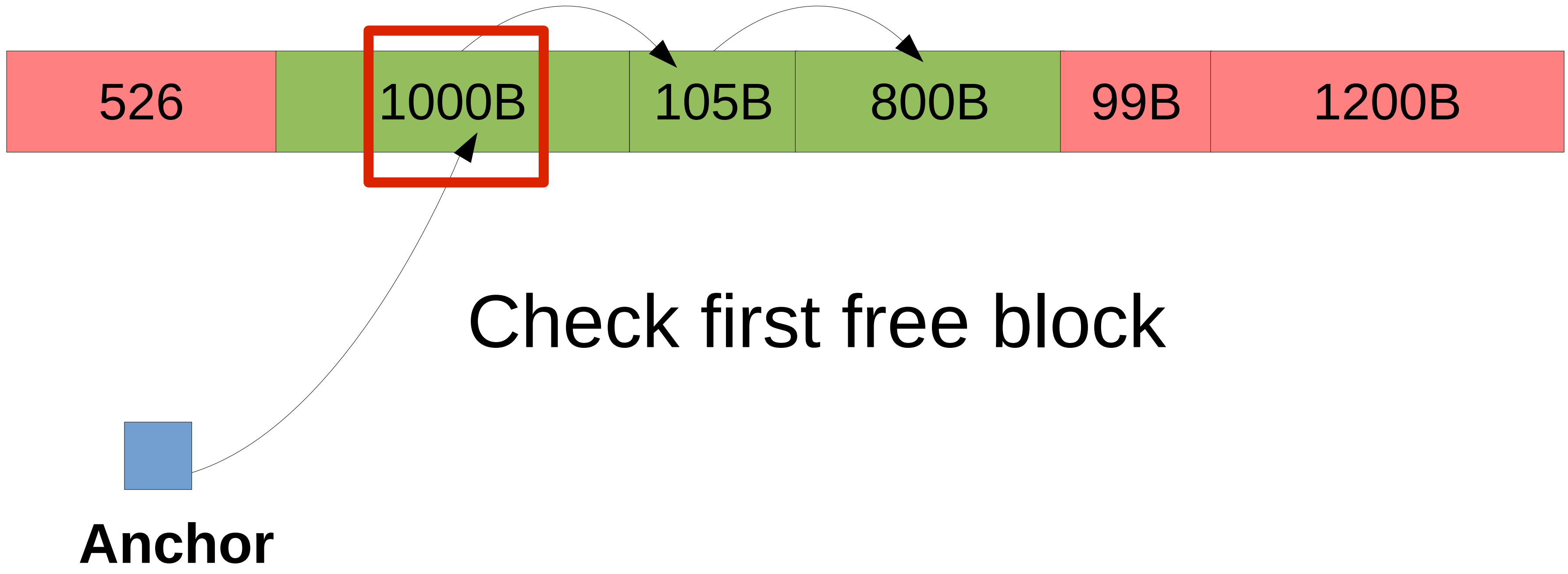
Building a Heap with Explicit Free List

Memory allocator: “**No Problem!**”



malloc(1200)

Building a Heap with Explicit Free List

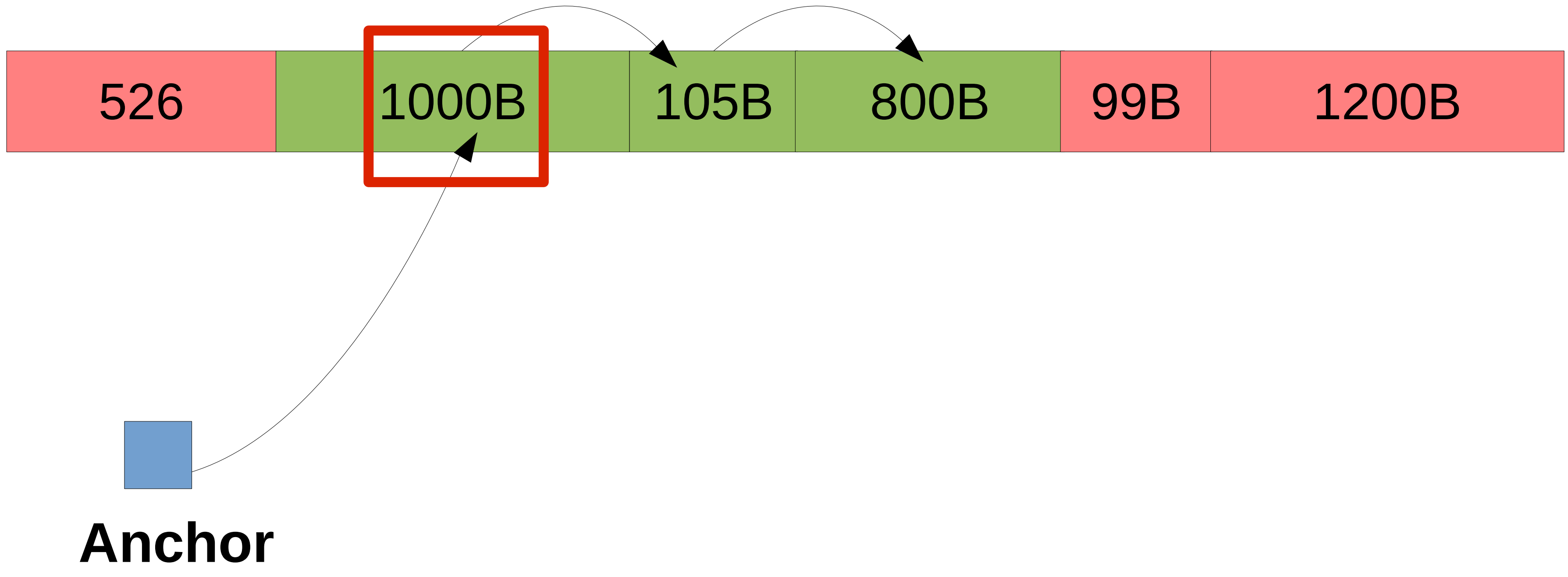


Check first free block

malloc(1200)

Building a Heap with Explicit Free List

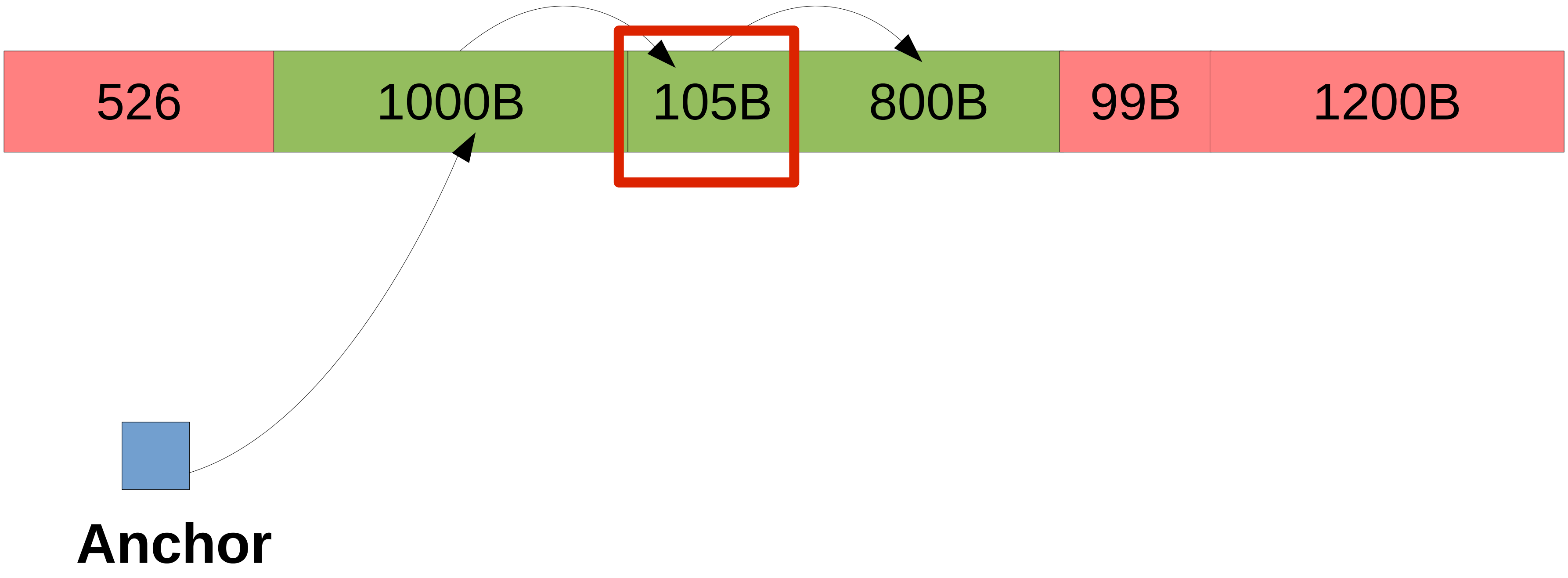
Hmmmm...not quite big enough



malloc(1200)

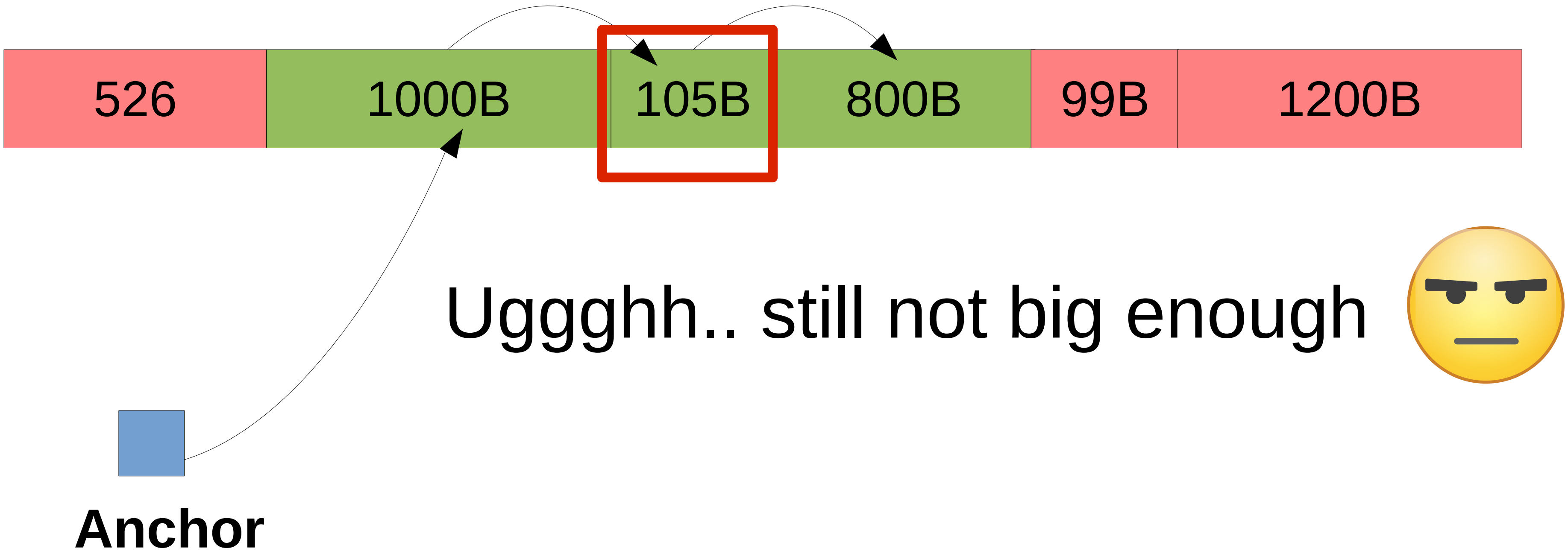
Building a Heap with Explicit Free List

No Problem! What about the next one?



malloc(1200)

Building a Heap with Explicit Free List

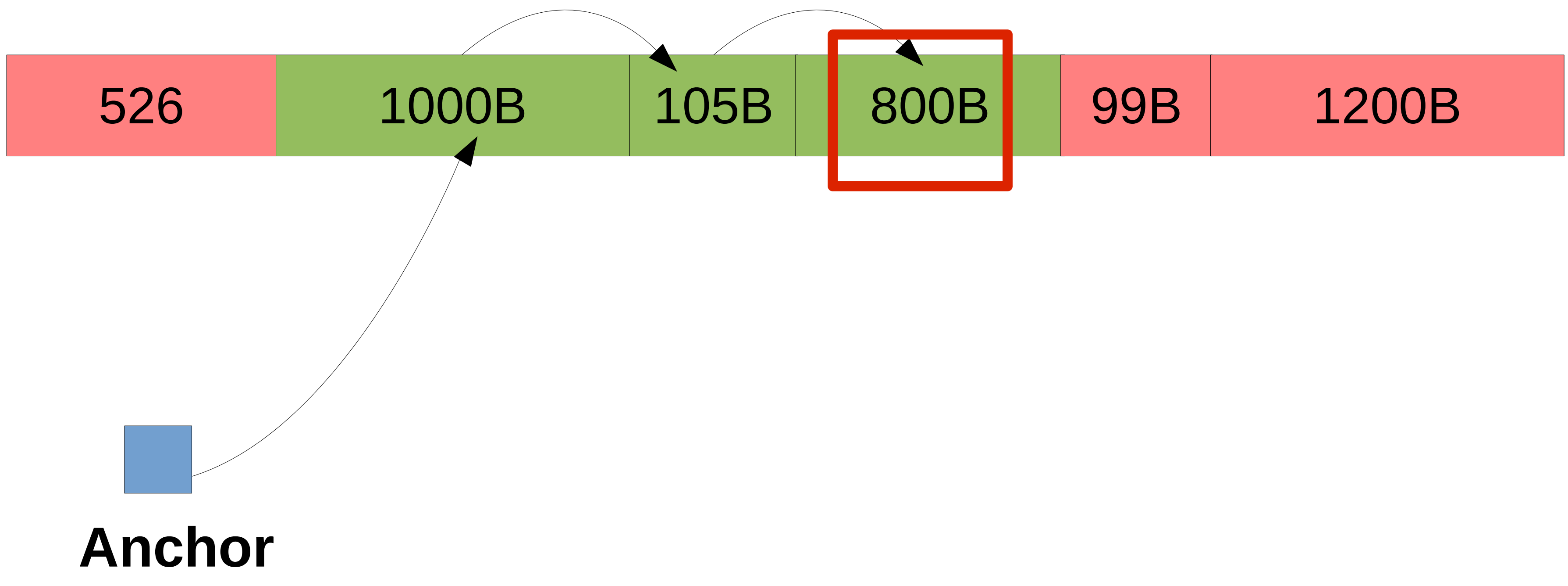


Uggghh.. still not big enough

malloc(1200)

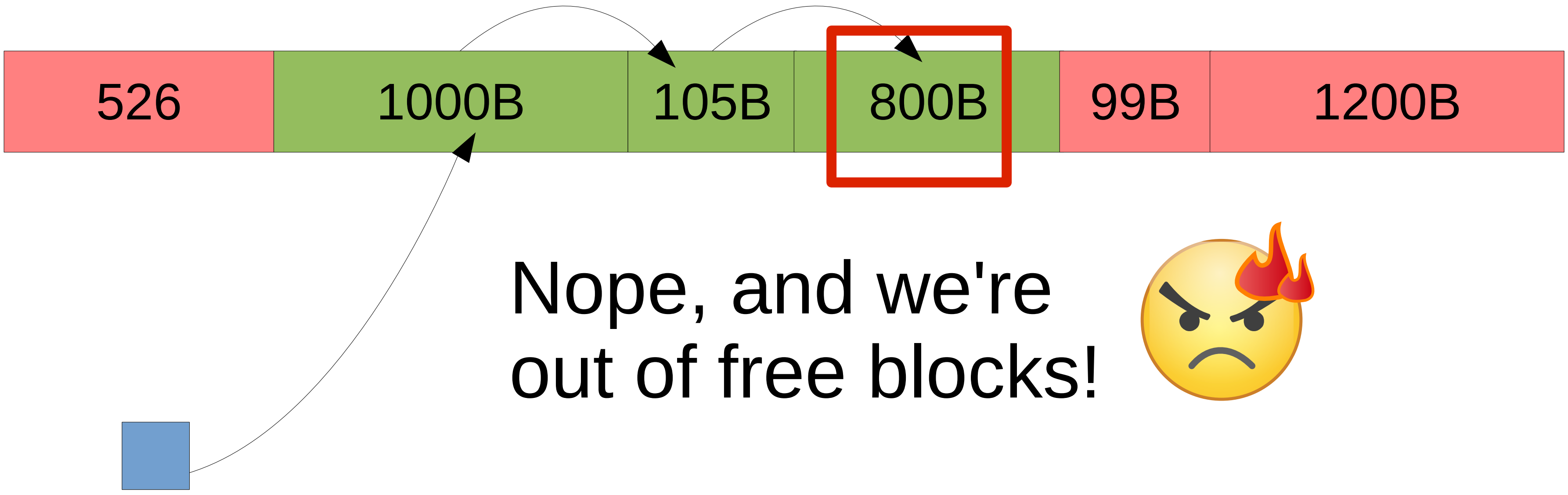
Building a Heap with Explicit Free List

OK, what about this one?



malloc(1200)

Building a Heap with Explicit Free List



Anchor

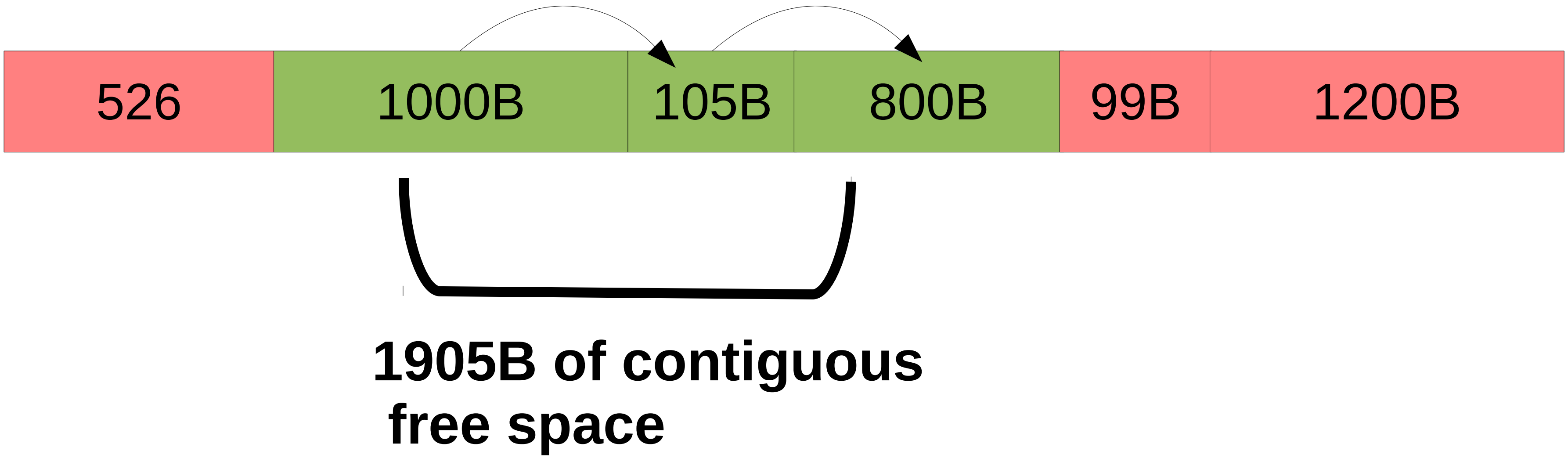
Nope, and we're
out of free blocks!

malloc(1200)

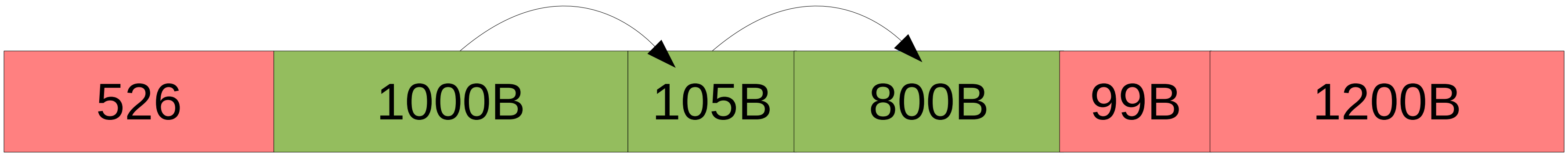
So what can we do?

Building a Heap with Explicit Free List

Well, there actually **is** enough contiguous memory to satisfy this request



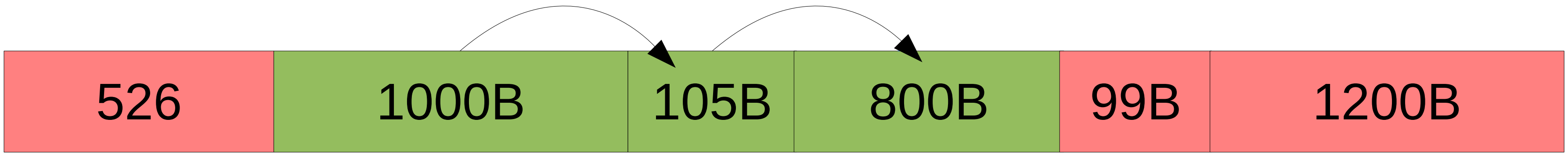
Building a Heap with Explicit Free List



But its fragmented into 3 smaller blocks...

So what should we do???

Building a Heap with Explicit Free List



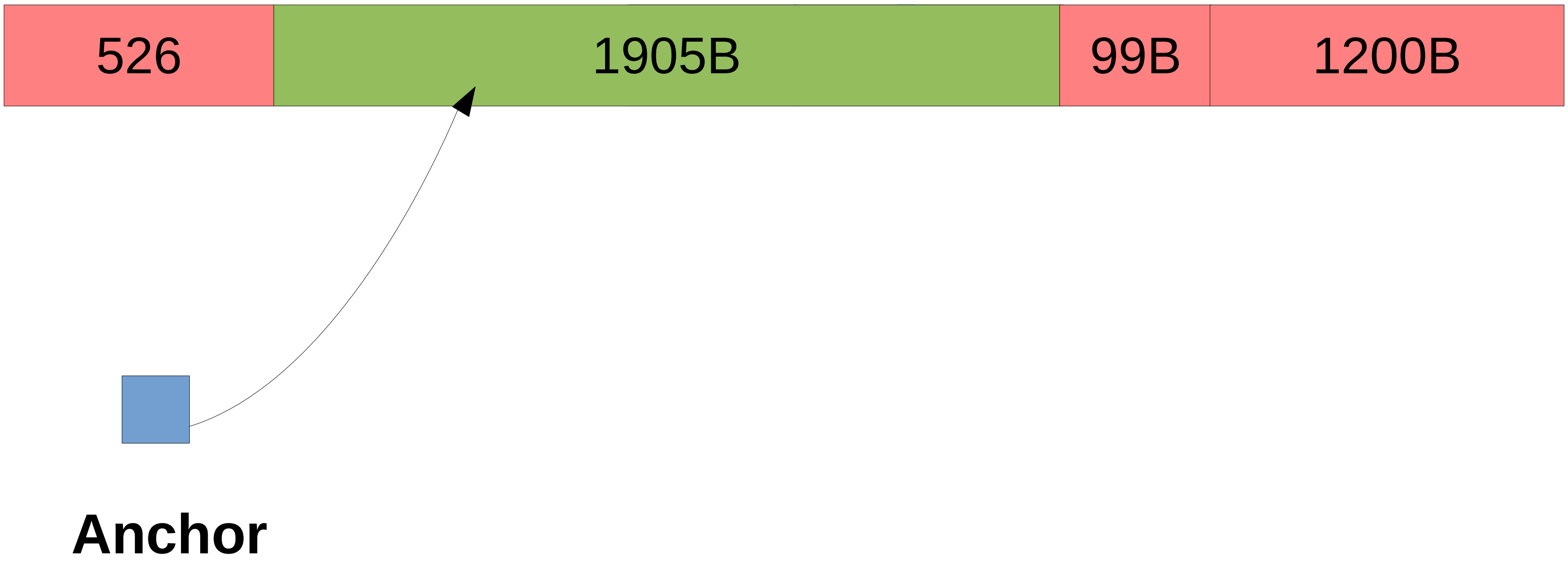
Yep, lets coalesce!

Immediate Coalescing

- Whenever we free a block, we immediately check whether this block can coalesce with either of its neighbors
- Also have to modify the free list

Building a Heap with Explicit Free List

This is the result



Useful Link

https://class.coursera.org/hwswinterface-002

The Hardware/Software Interface: video lectures from University of Washington