Memory Allocators

Asim Maharjan
Madhav Aryal

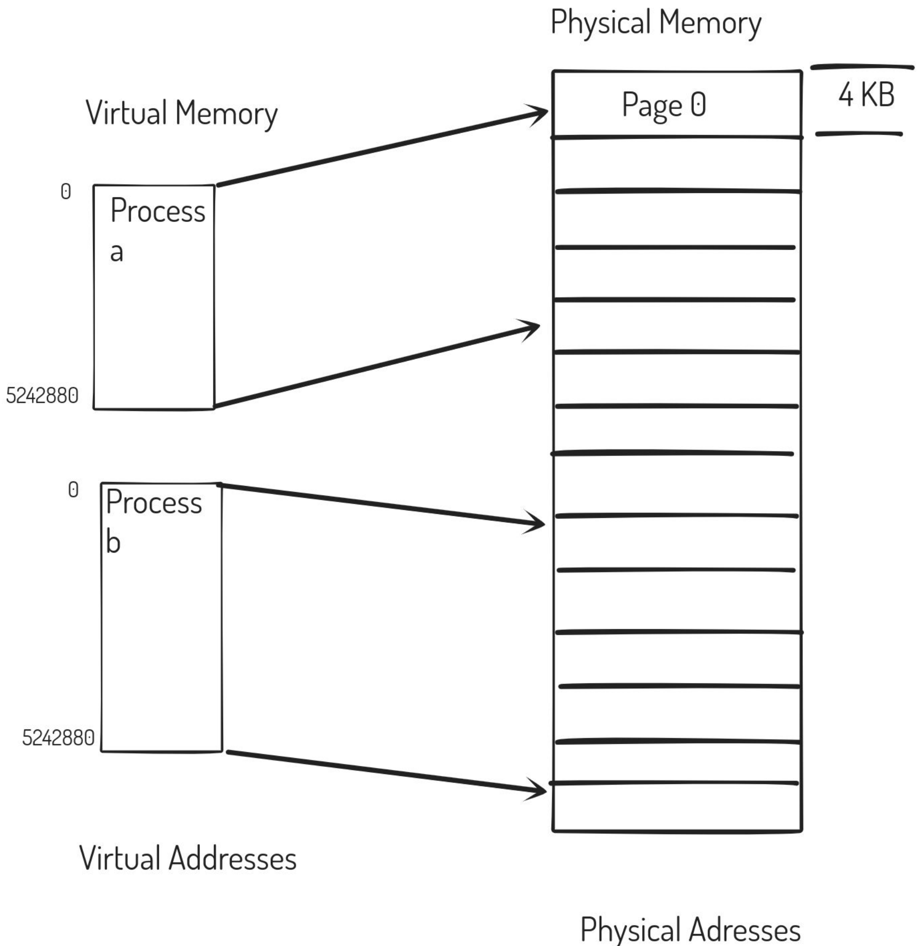
Revision

- Virtual Memory
- Pointer Arithmetic
- Bitwise Operations

Contents

- Allocating Virtual Memory (Windows and Linux)
- Memory Alignment
- Types of Allocators
    - Bump Allocator
    - Stack Allocator
    - Free list Allocator

Virtual Memory



Allocating/Obtaining Virtual Memory

- Memory is a resource managed by the Operating System
- Use system calls to request OS for more memory
- The OS will allocate pages, creates mappings and returns back a pointer to memory
- Need to utilize OS specific functions
    - *mmap* for Linux based systems
    - *VirtualAlloc* for Windows systems

Page Allocation

- OS will always allocated virtual memory in terms of pages.
- Consequence: Memory size allocated will always be a multiple of the Page size of the operating system
- Example, if page size of OS is 4kB, and we request a page having 2kB, then OS will return page of size 4kB regardless.

Memory Protections

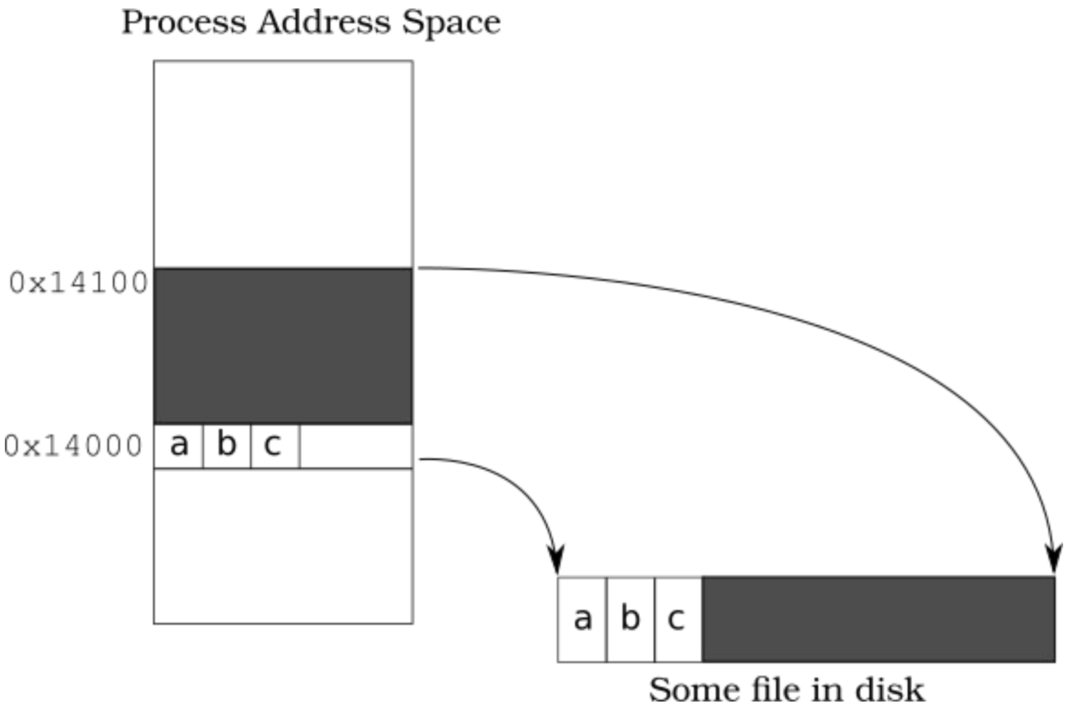
- Provided by the OS.
- Three common types of protections:
  - Read Protection
  - Write Protection
  - Execute Protection

The mmap system call

- Used in Linux based system
- Maps files or devices into memory i.e, creates a memory mapping
- Enables memory mapped I/O

Memory Mapped I/O

- I/O in which devices/files are accessed through memory addresses (pointers)
- Common example is file I/O which utilizes your HDD or SSD
- Any changes made to the memory through the pointer will also be reflected into the file.



Relation with Virtual Memory

- Virtual pages are stored in disk
- *mmap* allows to creates mappings to files on disk
- Creating mappings to virtual pages
- Named as **anonymous mapping**

Using mmap for Memory Allocation

```
void *mmap( void *addr, size_t length, int prot, int flags, int fd,
            off_t offset);
```

Returns a pointer to memory on success, MAP_FAILED on failure.

addr	Starting address of memory to allocate. If set to NULL, OS will return a suitable pointer instead.
length	Size of the memory in bytes we want to allocate.
prot	Protection flags
flags	Controls properties of the mapping
fd	Set to -1 for our purposes
offset	Set to 0 for our purposes

mmap: An Example

size of 4kB

Enable Read and write to allocated memory

addr set to null

```
size_t size = 4096;
void *mem = mmap( NULL, 4096, PROT_READ|PROT_WRITE,
                  MAP_ANON|MAP_PRIVATE, -1, 0 );

if ( mem == MAP_FAILED ){
  // .. error
  exit(1);
}
```

Create a private and anonymous map

fd and offset set to -1 and 0

The munmap system call

- Undos a mapping created by mmap
- Basically, deallocates the pages allocated with mmap

```
int munmap( void *addr, size_t length );
```

- Undos the mapping of all pages that are in address range of **addr** to **addr+length**
- Returns 0 on success and -1 on error.

mmap Full Example

```
size_t size = 4096;
void *mem = mmap( NULL, size, PROT_READ|PROT_WRITE,
                  MAP_ANON|MAP_PRIVATE, -1, 0 );

if ( mem == MAP_FAILED ){
  // .. error
  exit(1);
}

if ( munmap( mem, size ) == -1 ){
  // error again
}
```

The VirtualAlloc function

- Windows counterpart of mmap
- “Reserve, commit or change the state of the region of memory within the virtual address space of a specified process.”

States of Memory

- Memory in Windows can be in 3 states: Reserved, Committed and Free
- Reserved
  - Only the address range is reserved and thus cannot be used by others
  - No pages are allocated
- Committed
  - Pages are allocated
- Free
  - The default state
  - Available for allocation

The VirtualAlloc system call

```
LPVOID VirtualAlloc(
 [in, optional] LPVOID lpAddress,
  [in]           SIZE_T dwSize,
  [in]           DWORD  flAllocationType,
  [in]           DWORD  flProtect
);
```

- Needs header <Memoryapi.h>
- Returns an LPVOID type return value. LPVOID means Long Pointer to Void, i.e, void *
- Returns NULL when fails otherwise returns starting address of allocated memory

Side note on Windows Type Naming

LPVOID	Long Pointer to VOID type, i.e, a void *
SIZE_T	Equivalent to size_t
DWORD	A 32-bit unsigned integer
[in]	The parameter acts as an input to the function

The VirtualAlloc system call

```
LPVOID VirtualAlloc(
 [in, optional] LPVOID lpAddress,
  [in]           SIZE_T dwSize,
  [in]           DWORD  flAllocationType,
  [in]           DWORD  flProtect
);
```

lpAddress	Similar to that of mmap
dwSize	Size of memory to be allocated
flAllocationType	Set to MEM_RESERVE|MEM_COMMIT
flProtect	Protection Flags

VirtualAlloc: An Example

```
SIZE_T size = 4096;
void *mem = VirtualAlloc(NULL,size,
                         MEM_COMMIT|MEM_RESERVE,PAGE_READWRITE );
if ( mem == NULL ){
  //.... error
  exit(0);
}
// Success: Use mem
// ......
```

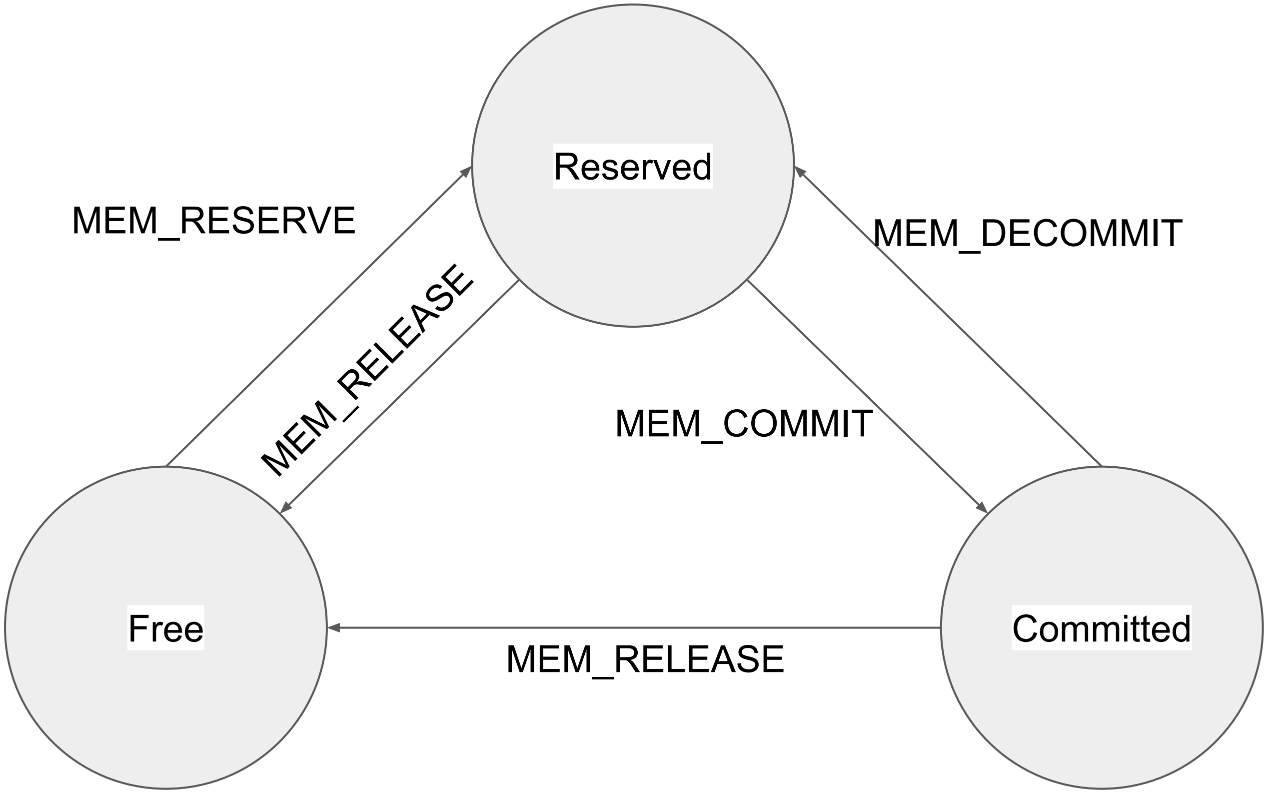
VirtualFree

- Decommits/Deallocates memory previously allocated by VirtualAlloc

```
BOOL VirtualFree(
  [in] LPVOID lpAddress,
  [in] SIZE_T dwSize,
  [in] DWORD  dwFreeType
);
```

lpAddress	Starting address of memory to be freed
dwSize	Size of memory to be freed
dwFreeType	MEM_DECOMMIT or MEM_RELEASE

States of Memory



MEM_DECOMMIT and MEM_RELEASE

- MEM_DECOMMIT sends pages back to MEM_RESERVED state
- MEM_RELEASE frees actual pages
    - dwSize **must** be 0 when MEM_RELEASE is used.

Virtual Free: An example

```
BOOL ret = VirtualFree(mem,0,MEM_RELEASE);
if ( !ret ){
  //... handle error
}
//.. No error
```

```
BOOL ret = VirtualFree(mem,4096,MEM_DECOMMIT);
if ( !ret ){
  //... handle error
}

ret = VirtualFree( mem, 0, MEM_RELEASE );
//.. No error
```

Memory Alignment

- A variable of a certain data type stored in memory follows an “alignment rule”
- The rule states which addresses are considered “valid” for a certain data type
- Some examples:
  - A 32-bit integer is always stored at an address that is a multiple of 4
  - A 64-bit integer is always stored at an address that is a multiple of 8
  - and so on..
- Dependent on the size of the data type.
- Access from an unaligned memory leads to slower code

Aligning to a Power of Two

```
uint64_t align_up( uint64_t size, uint64_t align ){
  return ( size + ( align - 1 ) ) &  ~(align-1);
}
```

- The above functions “aligns” the value of size to the power of 2 given by align
- align_up(23, 8) = 24
- align_up(27,8) = 32
- align_up(16,8) = 16

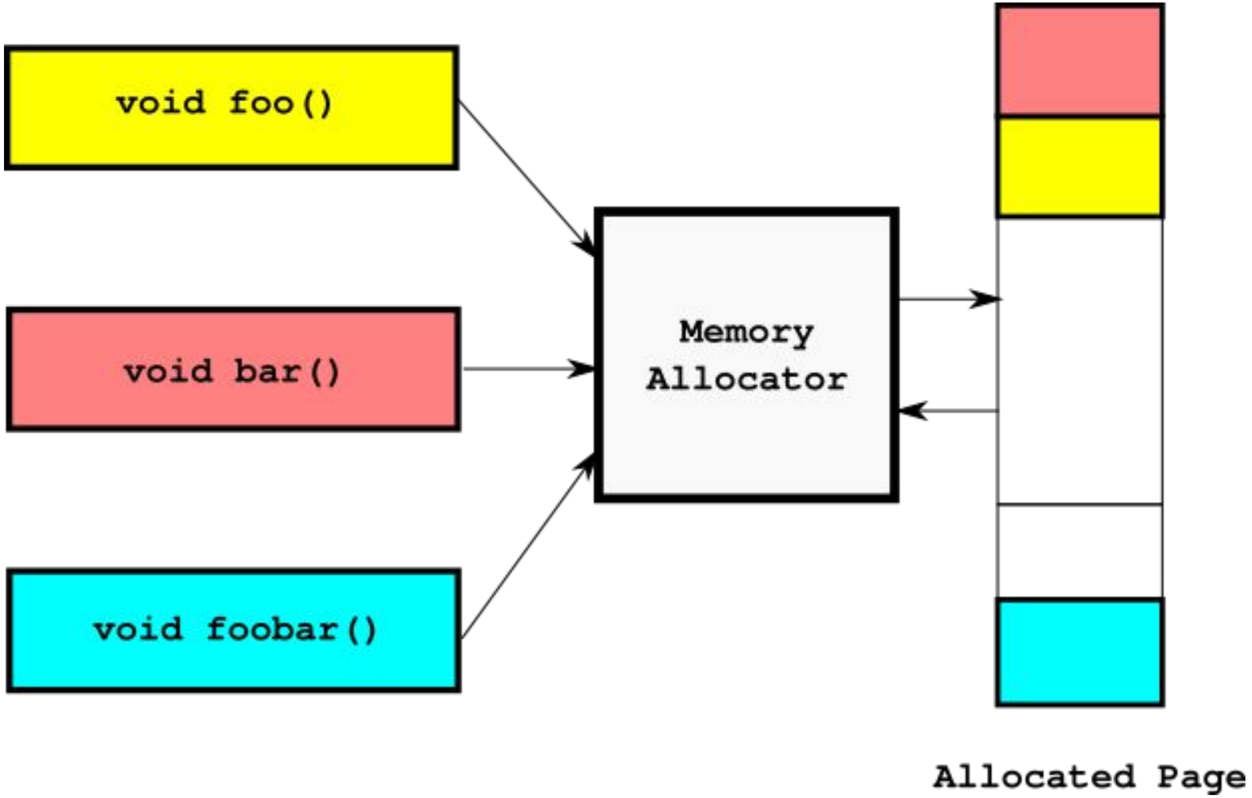
Aligning Up to a power of Two: Breakdown

- What does `X & ~(Y-1)` do when Y is a power of two?
  - Hint: Write Y-1 in binary
- What is the value of (X + (align-1))?
  - Check for different values of X, when X is a multiple of align and when it is not

BREAK

Memory Allocators

- After allocating virtual memory from the OS, we are left with a single big chunk of memory
- We still need some way to distribute and manage this chunk of memory among different parts of our program
- Main functionalities:
  - An ‘alloc’ function to allow our program to request memory
  - A ‘free’ function to return memory back so that it can be reused.
  - Allows for addresses of arbitary alignment


void foo()
void bar()
void foobar()
Memory Allocator
Allocated Page

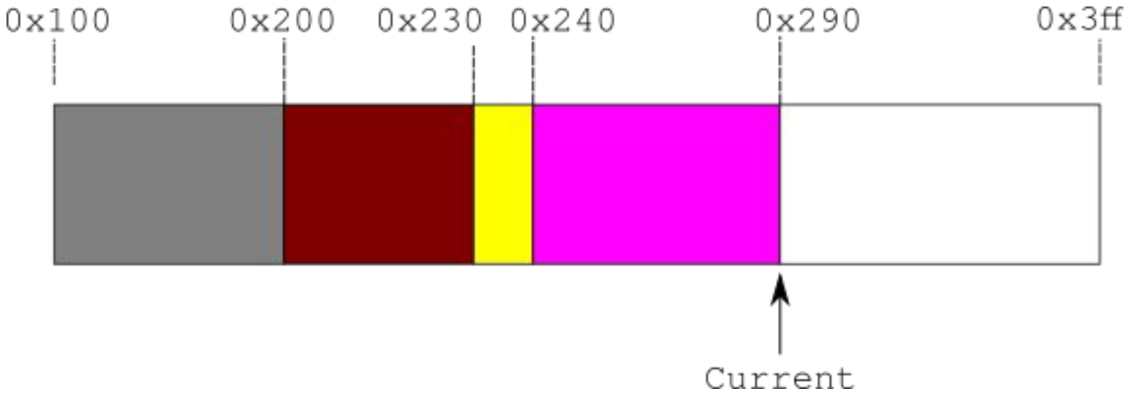
Memory Allocators: Types

- We discuss 4 different types of allocators
- In Increasing order of their complexity and flexibility:
  - Bump Allocator
  - Stack Allocator
  - Pooled style Free list Allocator
  - Generic Free List Allocator
- All allocators have their own way of performing memory allocation and deallocation

Bump Allocators

- Simplest kind of memory allocator
- Allows for memory allocations
- Allows limited freeing of memory
- It is **fast** and can be reused easily.

Bump Allocator: The Bigger Picture



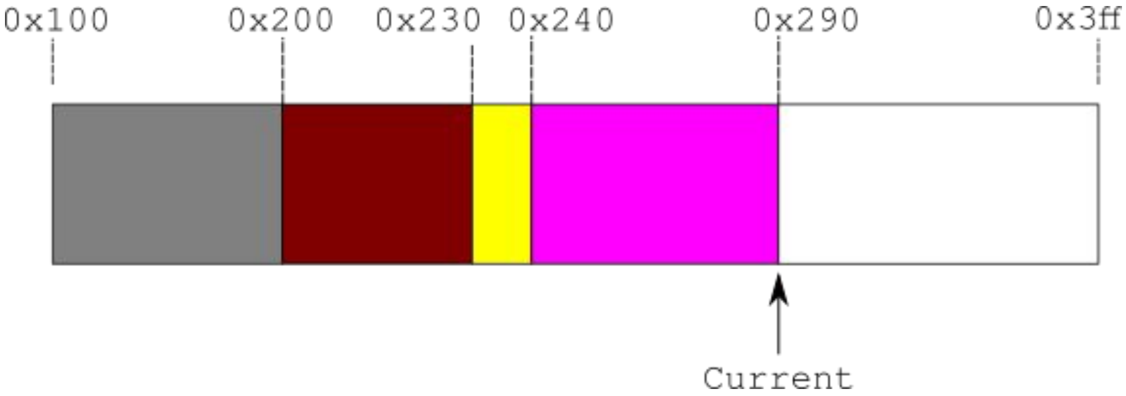
- Different color indicate memory allocated by different parts of the program
- Gray region has address of 0x100, Red(maroon) has address of 0x200 and so on.
- The next allocation made will have the memory address 0x290.
- Alignment is not considered (yet).

Bump Allocator: Data Structure

- **Current** pointer, which points to base of unallocated memory
- Pointer to the memory allocated from the OS.
- Size of the memory allocated from the OS.

Bump Allocator: Data Structure

- In our previous example:
  - **Current** = 0x290
  - Pointer to memory allocated from OS = 0x100
  - Size of allocated memory = 1024 bytes

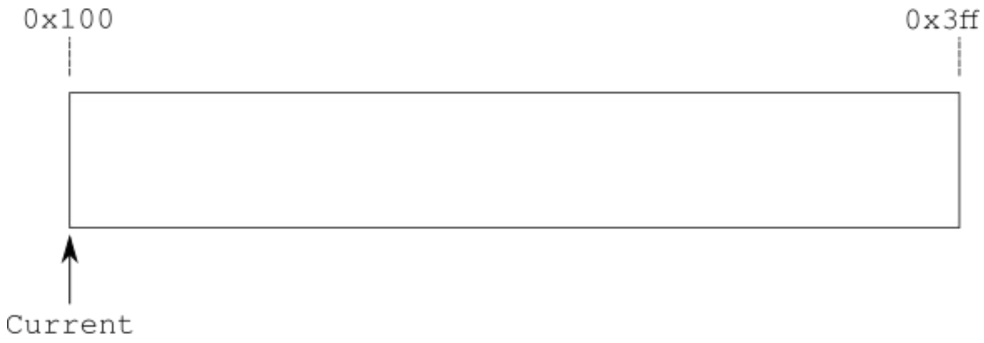


Bump Allocator: Allocation

- Consider allocation of X bytes of memory with a given alignment
- The steps involved include:
  - Aligning value of **current** pointer to the given alignment
  - Saving value of **current** pointer
  - Incrementing value of **current** pointer by X bytes
  - Performing bounds check
  - Returning the allocated memory (i.e, the save **current** pointer)
- Consider the allocation of 37 bytes initially, with an alignment of 8 bytes

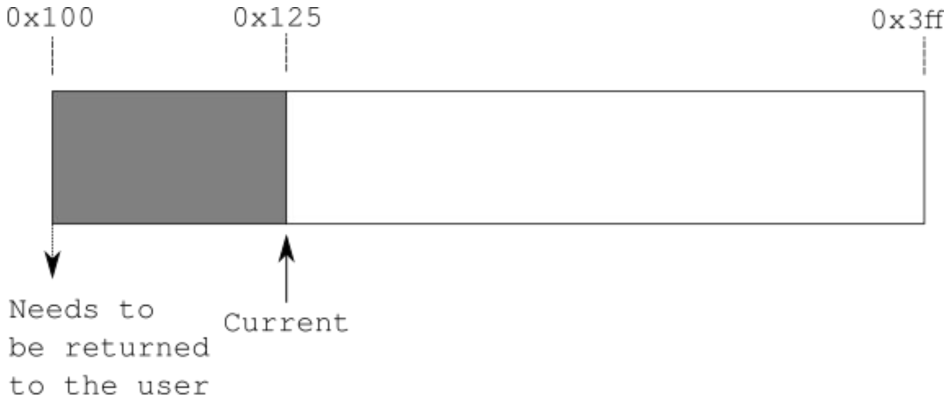
Bump Allocator: Allocation

- Initial State:
  - **Current** = 0x100
  - Memory pointer = 0x100
  - Size = 1024 bytes
- Notice that **current** pointer is already aligned to 8 bytes



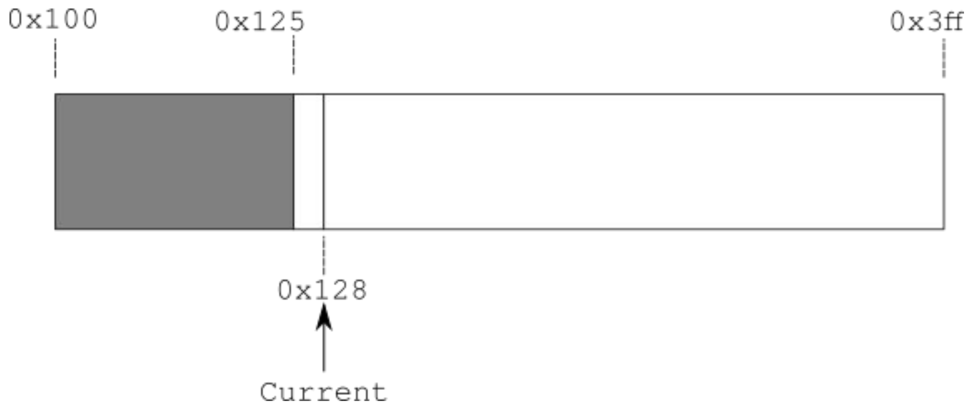
Bump Allocator: Allocation

- The **current** pointer is incremented by 37 bytes.
- Bounds checking the **current** pointer
  - Lies inside the allocated memory from the OS
  - Ensures that 37 bytes can be utilized
- The pointer value of 0x100 is returned



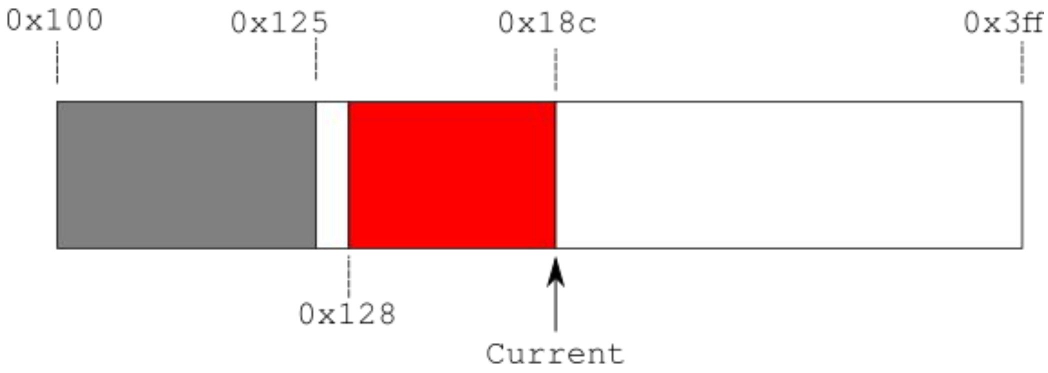
Bump Allocator: Second Allocation

- Now consider allocation of another 100 bytes with 8 byte alignment
- Align **current** pointer to 8 bytes
  - align(0x125,8) = 0x128



Bump Allocator: Second Allocation

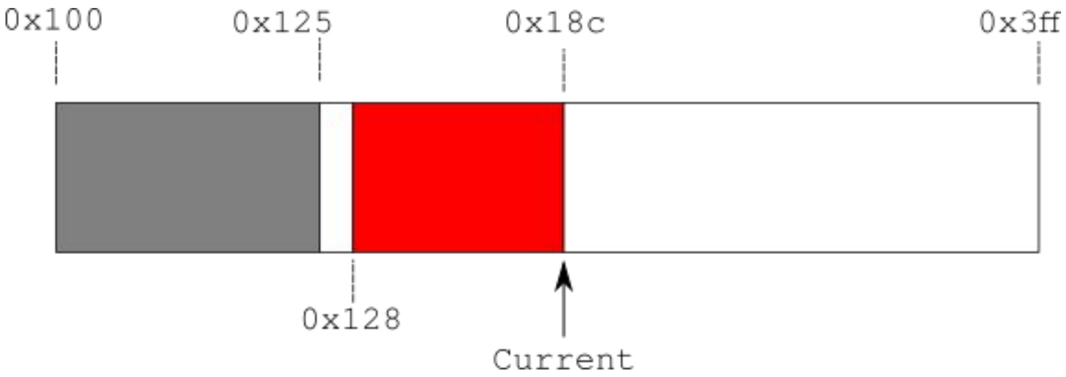
- Increment **current** pointer by 100 bytes
- Bounds checking the **current** pointer
  - Lies inside the allocated memory from the OS
- The pointer value of 0x128 is returned



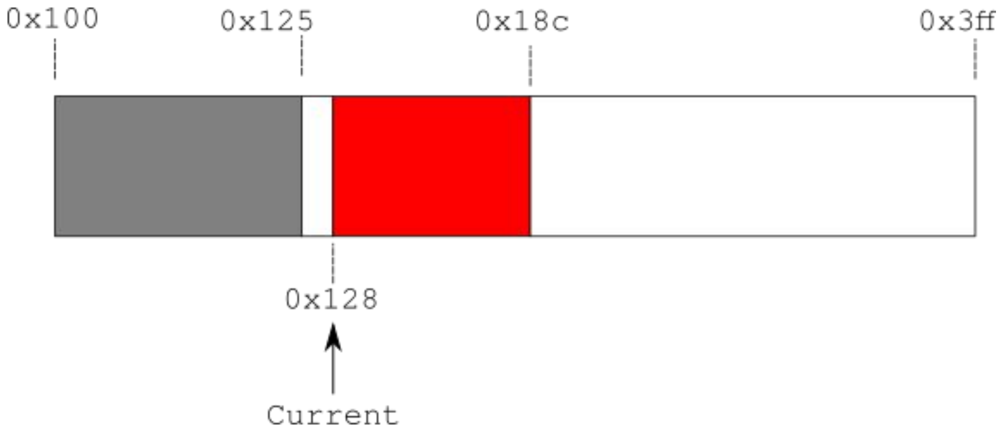
Bump Allocator: Freeing Memory

- Limited when it comes to freeing memory
- Can only free memory from the top
- Size of memory to be freed needs to be passed
- Freeing is simple
  - Simply decrement the **current** pointer by the size to be freed
  - Bounds checking still needs to be done

Bump Allocator: Freeing Memory Example



After Freeing by 100 bytes:



Bump Allocator: Resetting

- Instead of freeing from the top, a more common operation is resetting of the bump allocator
- Simply sets the allocator to its initial state
- Easily Re-use the allocator for other purposes

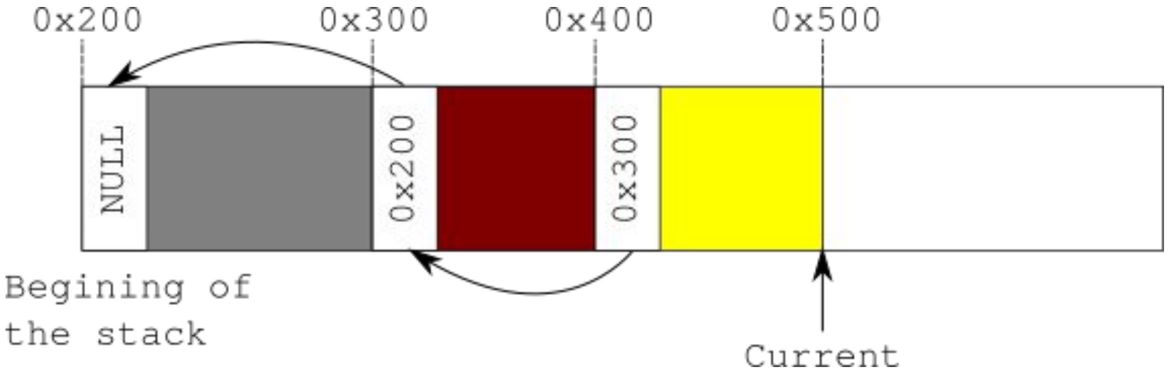
Bump Allocator: Summary

- Advantages:
  - Simple and Fast
- Drawbacks
  - Cannot grow in size
  - Limited freeing
- Useful for temporary memory, i.e, applications where we need memory only for a relatively small number of operations.

Stack Allocators

- Similar to a Bump Allocator in their operation
- However, it keeps track of every allocation
- Freeing however, still can only be done from the top

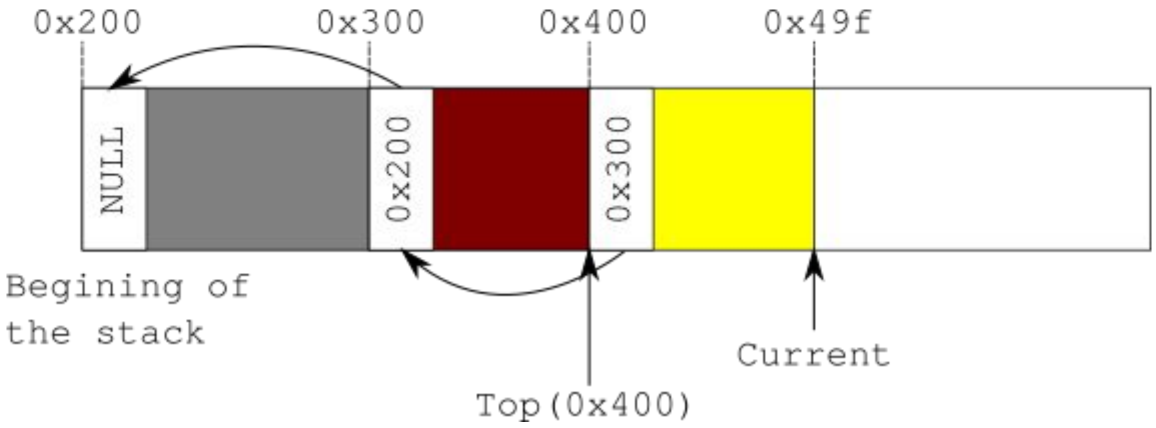
Stack Allocators: The Bigger Picture



- Allocations are tracked using a linked list of pointers that goes backwards

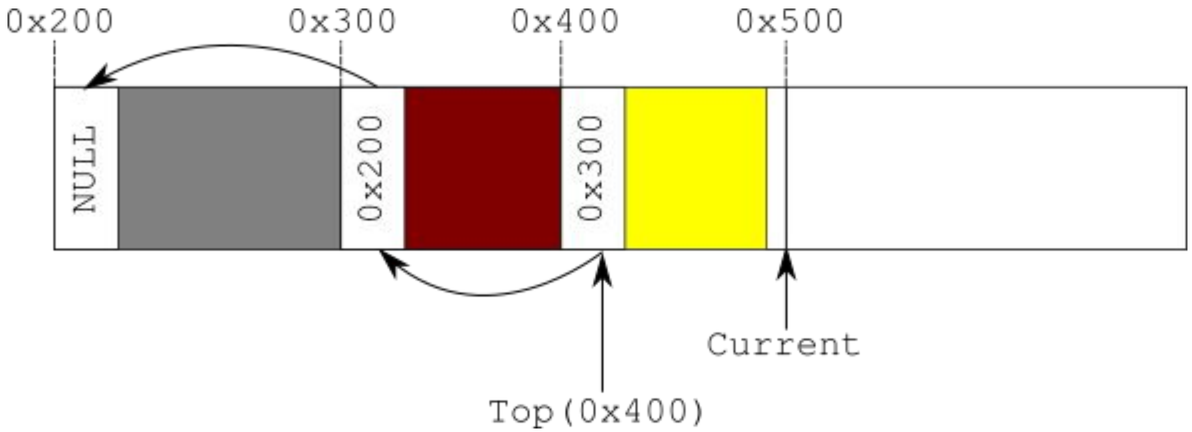
Stack Allocator: Data Structure

- **Current** pointer, which points to base of unallocated memory
- Pointer to the memory allocated from the OS.
- Size of the memory allocated from the OS.
- **Top** pointer, which points to the beginning of the linked list structure



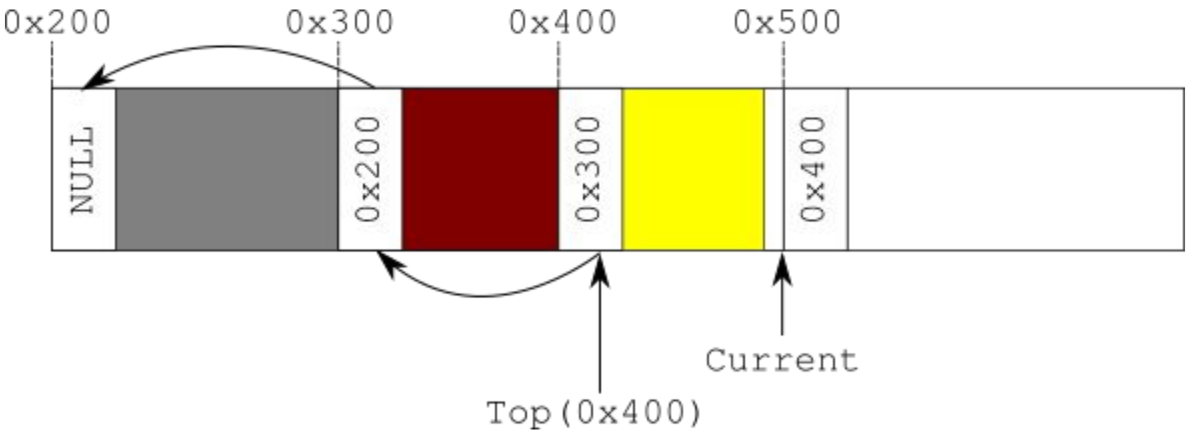
Stack Allocator: Allocation

1. Align the **current** point suitably so that we can store a pointer.



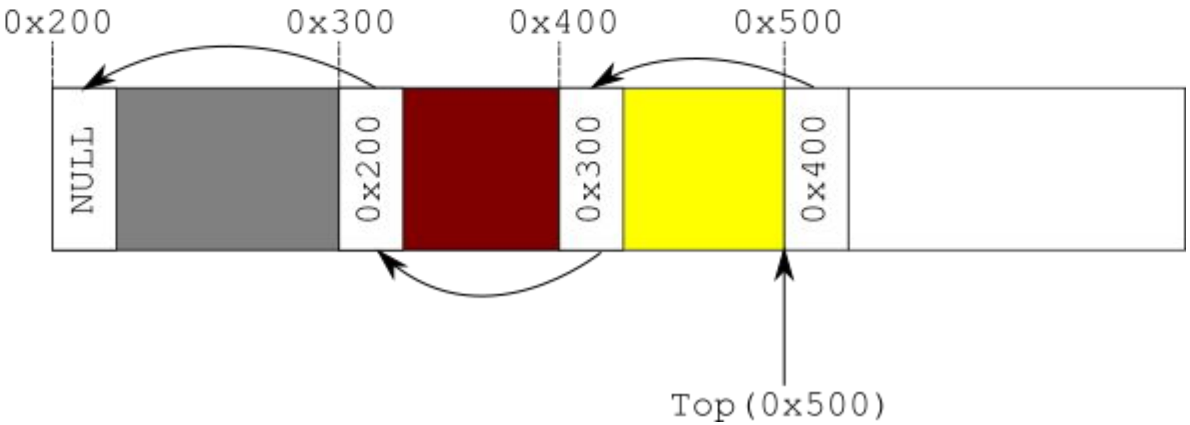
Stack Allocator: Allocation

2. Store the current **Top** value in that address



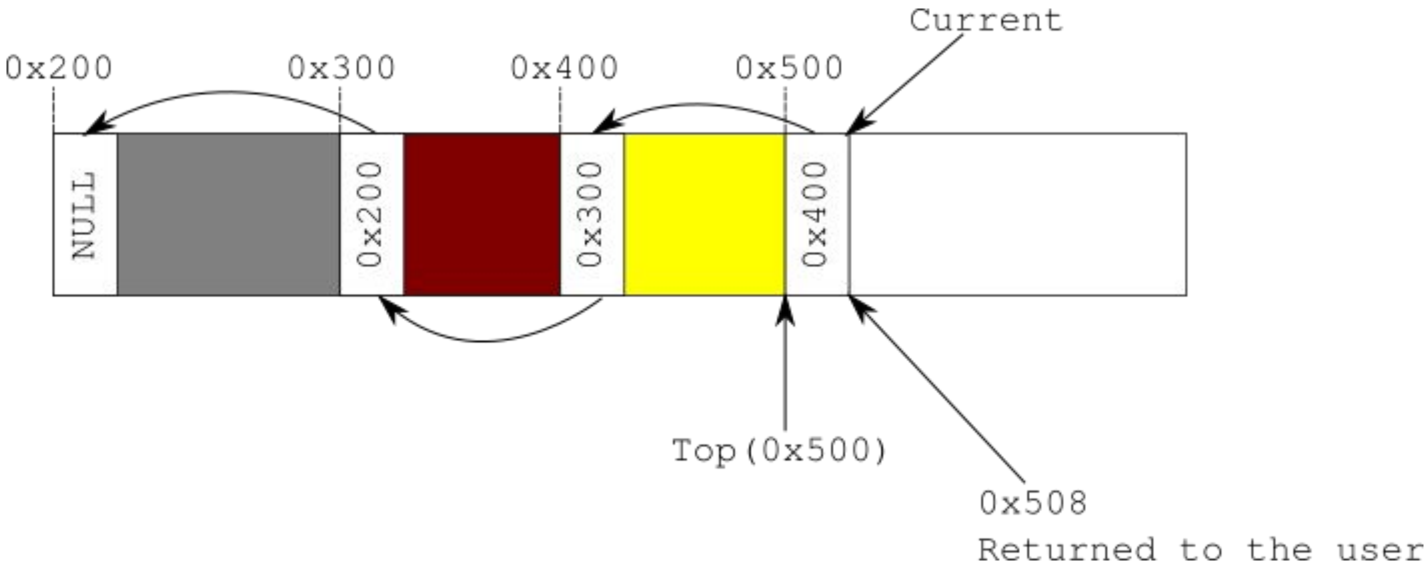
Stack Allocator: Allocation

3. Set the **Top** value to the **Current** pointer



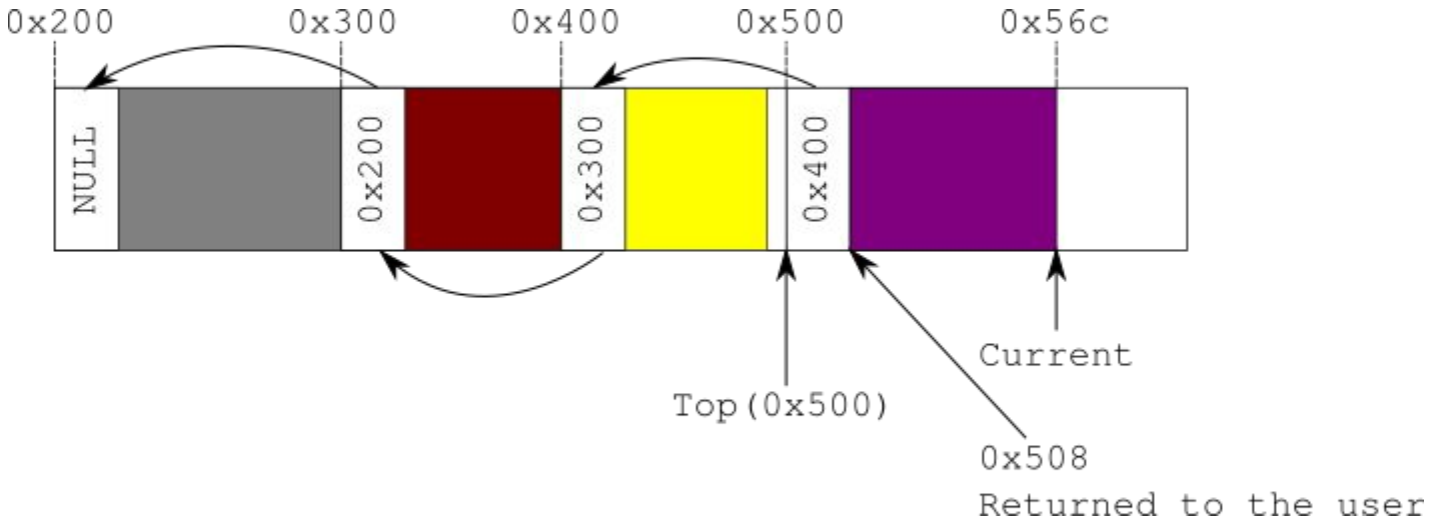
Stack Allocator: Allocation

4. Point to the next memory address after. This is the address to be returned back. Any required alignment should be done in this step.



Stack Allocator: Allocation

5. Allocate the required number of bytes as done in bump allocator.

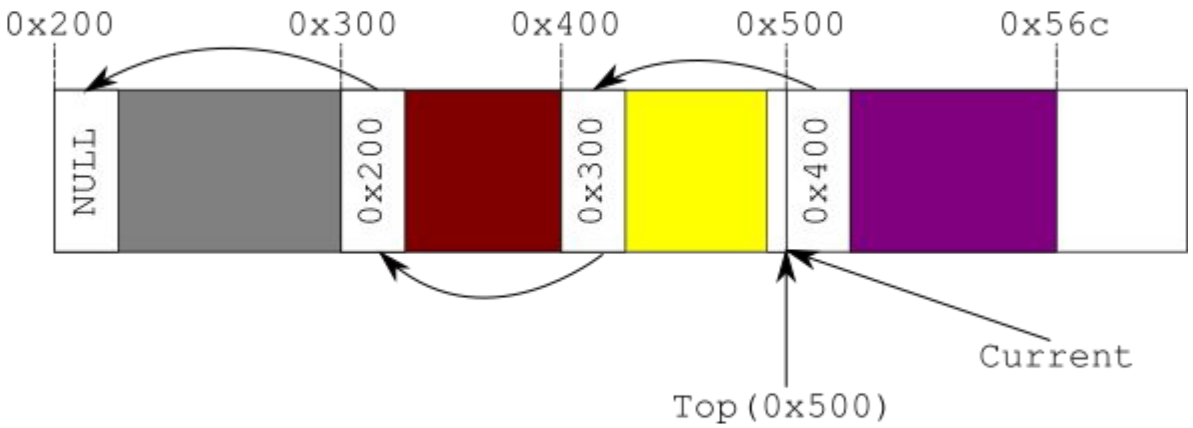


Stack Allocator: Freeing Memory

- Stack allocator provides a **Pop()** function that can be used to free memory at the top
- The **Pop()** does not need a size argument to be passed

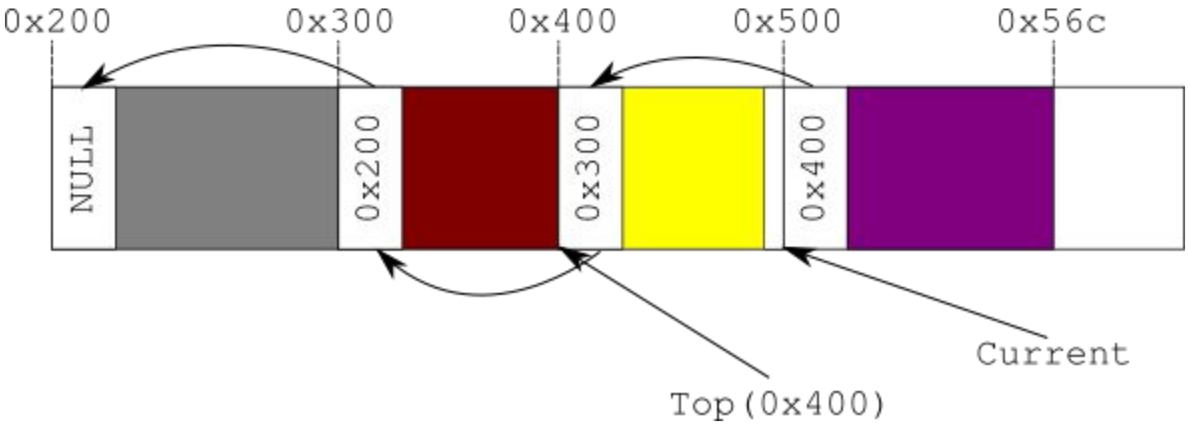
Stack Allocator: Freeing Memory

1. Set **Current = Top**



Stack Allocator: Freeing Memory

2. Set **Top = *Top**



Stack Allocator: Summary

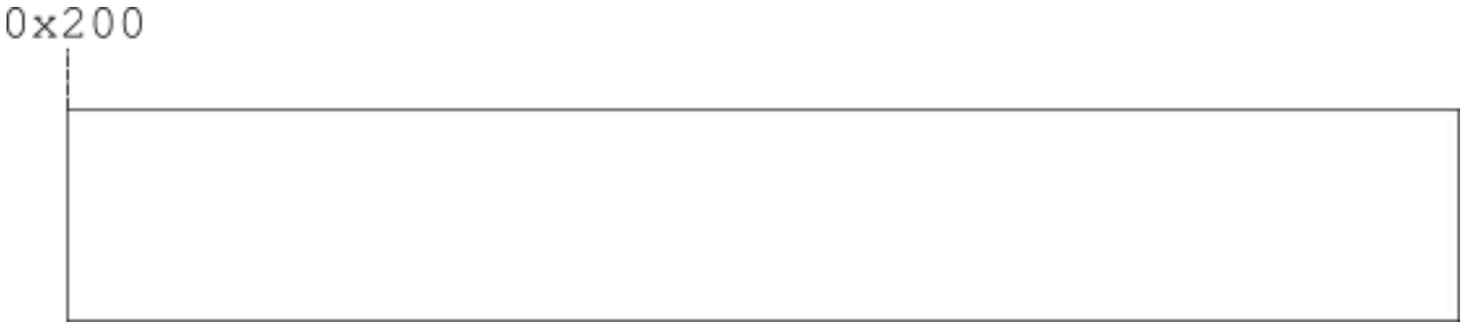
- Not very useful compared to bump allocator
- Added a lot complexity for addition of a very small functionality
- Total size of allocator is fixed.

Free List Allocator

- Uses a linked list to store memory chunks
- Memory chunks are of fixed size
- Allocations of only a fixed size can occur using this kind of free list
- Allows for general freeing of memory blocks
- We do NOT consider the case for an arbitrary aligned memory.

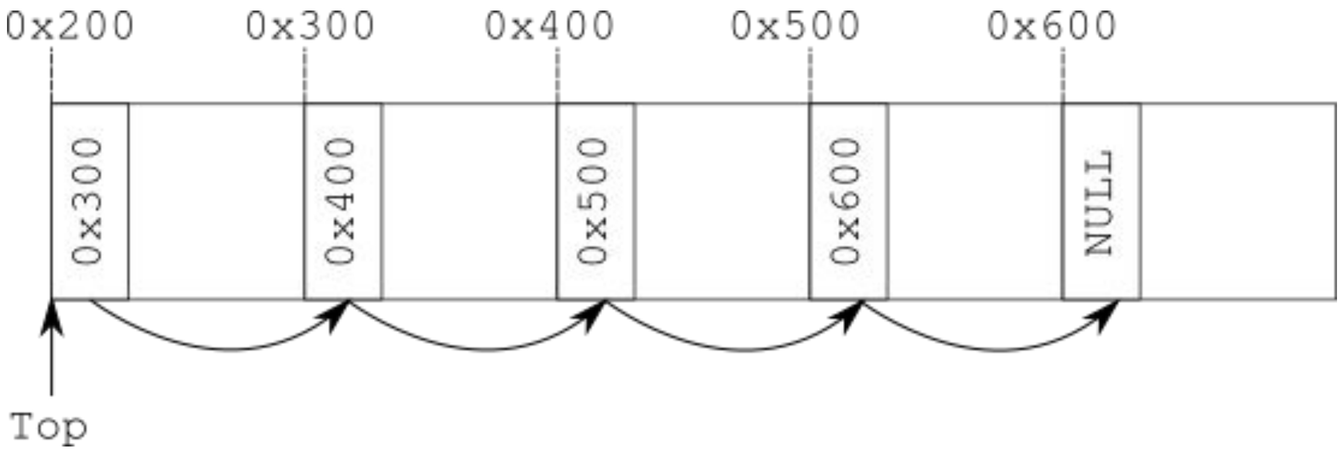
Free List Allocator: Algorithm

1. Start with an empty page



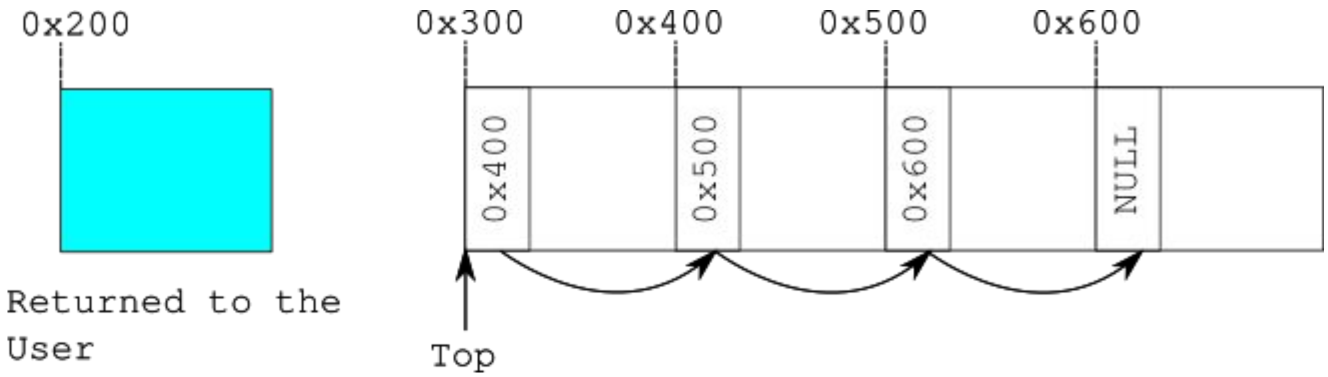
Free List Allocator: Algorithm

2. Create a linked list of memory chunks. Here, we have assumed a chunk size of 256 (0x100) bytes.



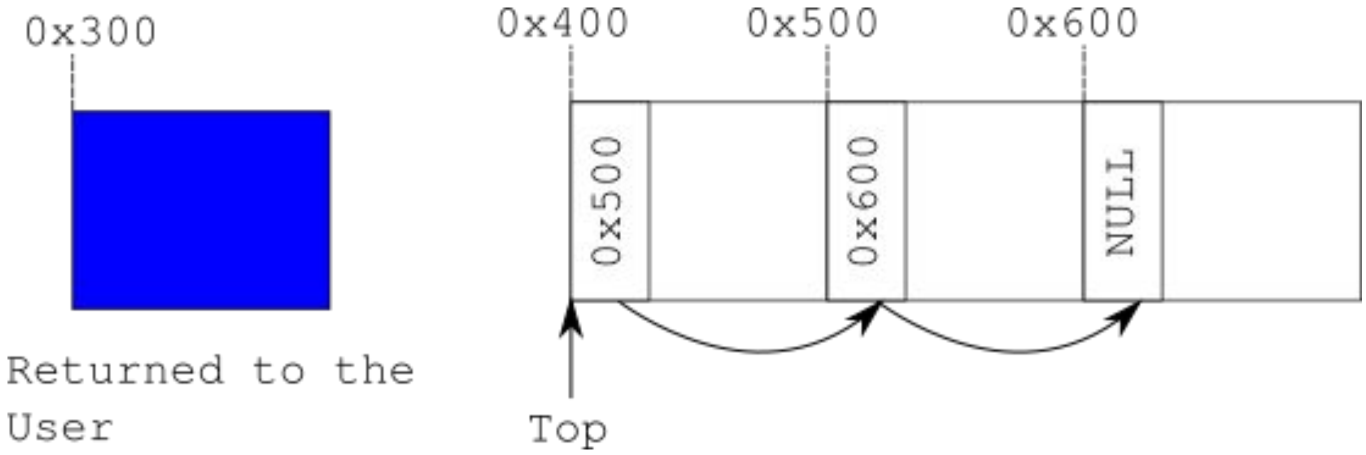
Free List Allocator: Allocation

3. To allocate a chunk, remove the Top node from the linked list and return it.



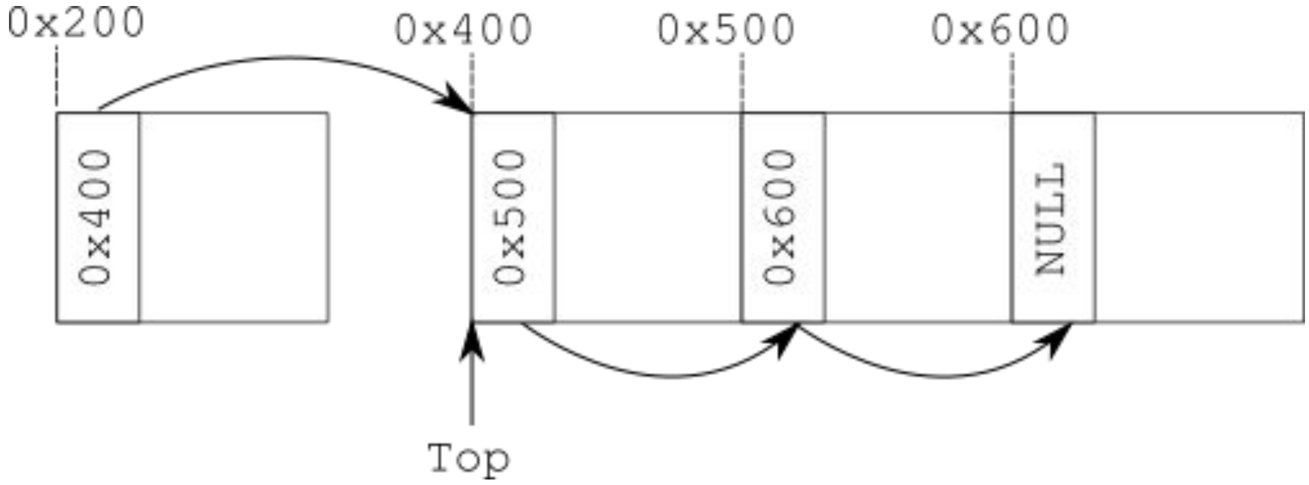
Free List Allocator: Allocation

Here's another example:



Free List Allocator: Freeing Memory

4. To free memory, we simply insert it into the HEAD of the linked list and store the pointer to the next chunk.



Free List Allocator: Getting New Memory

- What happens when free list runs out of chunks?
  - Simply allocate more Virtual Memory and perform the same steps as above
- Need to keep track of all the virtual memory allocated so far
  - Required to free all the created mappings

Free List Allocator: Summary

- Allows for freeing of individual allocations
- Can grow in size
- Drawbacks
    - Allocation size is fixed