System Allocation Functions:

Recall:

Virtual Memory

Memory Allocation:

Memory is a resource managed by the OS. So, whenever we want to use more memory, we have to request the OS for more memory. Operating Systems provides functions through which we can make this request. Also, remember, any memory the OS provides us with will be **virtual** i.e, the contents of memory that we will be using will be stored in virtual pages and will be swapped in/out whenever they are used/unused.

In modern OSes, each process has its own address space, distinct from the address space of other processes. So, each process has a memory address (theoretically) ranging from 0x0000000000000000 - 0xffffffffffffffff. At the beginning however, these memory addresses do not actually "map" to physical memory. To actually use the physical memory (RAM), it is necessary to create a mapping between the memory address of the process and the actual physical memory. This mapping is created by the OS. To request the creation of such a mapping, and thus to "allocate" memory, we have to use various system calls (provided by the OSes).
Using a memory address that has not been "mapped" will result in what is commonly known as "Segmentation Fault".

Here, we will look at VirtualAlloc for Windows and mmap for Linux based OSes which are used to create mapping between a process's address space and the physical memory.

When we create a mapping it is also necessary to delete them, i.e, to “free” the memory. This process is called “deallocation”. It is done by VirtualFree and munmap system calls.

Memory Protection:

OSes also allow programmers to control read/write and execution of memory. The three common memory protections are:

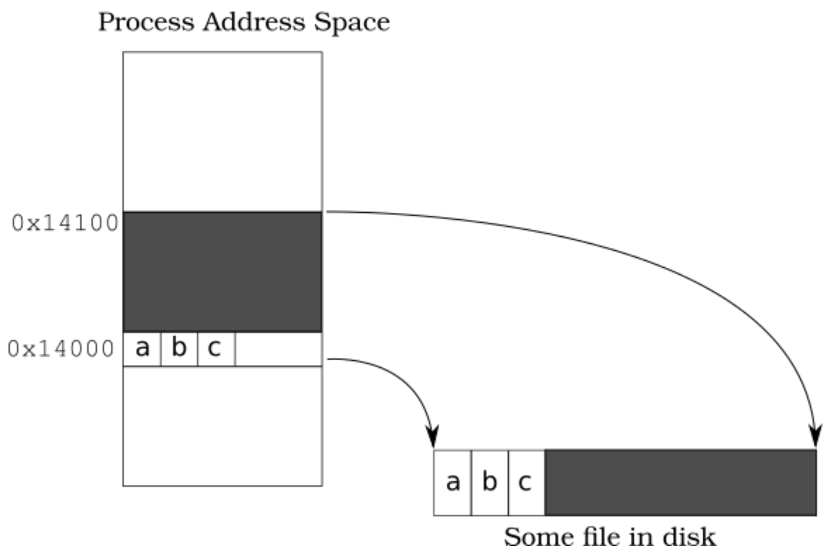
- Read Protection
  Controls whether the memory can be read from or not.
- Write Protection
  Controls whether the memory can be written to or not.
- Execution Protection
  Controls whether the contents of the memory can be executed or not. Since compiled code is just binary data, it can be loaded into memory location with execution permission and then executed.

Violating memory protection will also result in Segmentation Fault from the OS.

mmap

mmap man page
The Linux Programming Interface (Book) - Chapter 49: Memory Mappings

The POSIX/Linux way of creating “memory mappings” in the virtual address space of our programs. The point of creating memory mappings is to make it so that the pointers/memory used in our program will actually correspond to some kind of resource (e.g. a virtual memory page or a file). In other words, memory mappings give a certain “meaning” to pointers.
These mappings can be either **file mappings** or **anonymous mappings**.



File Mappings are used to map files directly into memory. Here, mapping is more akin to "loading" a file directly into the memory. However, any changes made to the contents of that range of "mapped" memory also appear on the file.

Anonymous Mappings are file mappings that don't correspond to files in the disk. It basically is a file mapping but without an explicitly defined file. The file in an anonymous mapping can be thought of as the virtual memory pages. Here, we will only briefly discuss anonymous maps, as they are used to allocate memory from the OS.

The mmap system call

The function signature of mmap is:

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

addr	The starting address of the memory you want to allocate. If addr is NULL, the OS itself will allocate a suitable memory address. Generally, this argument will be NULL.
length	Length in bytes of the memory you want allocated

prot	Memory protection flags. Can be used to restrict/grant read/write/execute access to the allocated memory.
flags	For our purposes, only MAP_PRIVATE and MAP_ANONYMOUS are relevant.
fd	File descriptor. -1 for our purposes.
offset	Set to 0 for our purposes.

Return Value:
On success, it returns the starting address of the memory map. Note that this may not be the same as the addr passed to the mmap call. On failure, returns MAP_FAILED.

The munmap system call:

Deletes the mapping created by mmap. The function signature is:

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

addr	The address returned by the mmap call.
length	The length of the mapping created.

Sample call to mmap:

```
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);
}

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

VirtualAlloc

VirtualAlloc - Documentation
VirtualFree - Documentation
Memory Protection Constants (for flProtect) - Documentation
Win32 Types - Documentation

"Reserve, commit or change the state of the region of memory within the virtual address space of a specified process."
The system calls to request memory / create memory mapping for your program in Windows.
The **VirtualAlloc** system call:

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

`LPVOID lpAddress`	Starting address of the memory to be allocated. If the parameter is NULL, the OS itself will choose a suitable memory address.
`SIZE_T dwSize`	Size of the memory to be allocated.
`DWORD  flAllocationType`	The type of operation to be performed on the memory address range specified by the lpAddress parameter. See below for more information.
`DWORD  flProtect`	The protection flags for the memory. Some common values would be PAGE_READWRITE, PAGE_READONLY, PAGE_EXECUTE etc.

VirtualAlloc can be used to perform following operations on memory. The operation is controlled by the flAllocationType parameter which can have different

values. The two more important operations are Reserving and Committing Memory.
To reserve memory, we call VirtualAlloc using the MEM_RESERVE value in the flAllocationType parameter. Reserving a memory address will not actually create pages of virtual memory, i.e, the mapping between the memory address and physical memory is not created. An address range which has been reserved cannot be reserved again.

Committing a memory address range will actually create the mapping between the address and the physical memory. Virtual pages will be created. However, pages will only be created when the memory will be first accessed, i.e, if the memory has not yet been accessed, pages will not be allocated. Committing is done by passing the MEM_COMMIT flag to the flAllocationType parameter in the VirtualAlloc function.

Reserving and committing can be done in one call to VirtualAlloc.

VirtualFree

Decommits or deallocates the memory that was previously allocated by a VirtualAlloc. The VirtualFree system call is:

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

`LPVOID lpAddress`	The starting address of the memory to be deallocated.
`SIZE_T dwSize`	Size of the memory region to be free.
`DWORD  dwFreeType`	MEM_DECOMMIT or MEM_RELEASE. See below.

Using MEM_DECOMMIT will send the pages back into a MEM_RESERVED state while using MEM_RELEASE will actually free the pages. MEM_DECOMMIT can be used to decommit **some** of the pages allocated by VirtualAlloc. To decommit pages, the value of lpAddress must be the base address of one of the pages. The value of dwSize determines the pages that will be decommitted. All the pages that lie within dwSize from lpAddress will be decommitted.

However, to **actually free** those pages, we must free all the pages allocated using VirtualAlloc at once by using MEM_RELEASE. To free all the allocated pages, the lpAddress must be the address that was returned by VirtualAlloc and dwSize must be 0.

Memory Usage

Memory Alignment

All the data stored in memory follows an “alignment rule”. The alignment rule dictates the way data is arranged in memory. For example, some of the alignment rules are:

- A 32-bit integer is 4-byte aligned, i.e, a 32-bit integer will be stored in a memory address that is a multiple of 4.
- A 64-bit integer is 8-byte aligned, i.e, a 64-bit integer will be stored in a memory address that is a multiple of 8.
- A 1-byte character will be 1-byte aligned, i.e, a character can be stored in any address.

Memory alignment is necessary to prevent multiple loads from RAM whenever we are accessing a variable. The CPU loads data a **memory word** at a time. The size of the memory word can be multiple bytes, usually a power of 2 like 32-bytes or 64-bytes or even more. The CPU, therefore, works with “memory word addresses” that are a multiple of size of the memory word.

So, for example, if we have a memory word of 32-bytes and we are accessing a 32-bit integer stored at an unaligned address, suppose 0x00003, then we would need two memory loads to read a single 32-bit integer. One memory load would be to load contents of 0x00000 and the other load would load contents of 0x00004.

So, we must make sure that our data is aligned properly to avoid unnecessary loads. Therefore, it is of utmost importance in memory allocators that the users are getting properly aligned memory according to their needs. So, we will have our memory allocation function take an extra alignment parameter as well.

Memory address is always aligned to the next multiple of the alignment size. The alignment for an unaligned pointer is carried out by following procedure.

```
void* Align( void *MemAddress, AlignSize ){
     MemAddress += (AlignSize - 1)
     MemAddress  = MemAddress & ~( AlignSize - 1 )
     return MemAddress
}
```

Data Types for handling Pointers

C provides us with special data types, mainly, intptr_t, uintptr_t and ptrdiff_t to handle arithmetic with pointers. A brief overview of these data types are:

intptr_t	Stores pointer as a **signed** integer.
uintptr_t	Stores pointer as an **unsigned** integer.
ptrdiff_t	Stores the difference between pointers as a signed integer.

The data types intptr_t and uintptr_t are always guaranteed to store all kinds of pointers. Primarily, uintptr_t is used for representation of a pointer as an integer. So, anytime there is a need to perform other kinds of arithmetic operations (e.g bitwise AND) on a pointer, be sure to convert it to a uintptr_t. And similarly, store the results of difference of pointers in a ptrdiff_t variable.

Memory Allocators:

Functions which provide memory to parts of programs/other functions. Generally, it is not feasible to make a system call (mmap/VirtualAlloc) whenever we need memory for our functions. System calls are slow and add unnecessary overhead to our programs. Thus, we need some kind of middle-man to manage the memory. Memory Allocator works in 3 simple steps:

1. Request a large chunk of memory from the OS.
2. Distribute this large chunk of memory optimally among the consumers/functions.
3. Deallocate the chunk of memory when everything is done.

The “optimal” part of distributing memory largely depends on the memory usage pattern of the program. This is the part of memory allocation that gives rise to **many** different memory allocation schemes, each with its own advantages and disadvantages.

A memory allocator should **at least** provide 2 functions, one for allocating the memory and one for freeing the acquired memory. We will call these functions Alloc() and Free() respectively. The functions signatures will generally be:

```
void *Alloc( size_t size, size_t alignment )
void *Free( void *ptr ) OR void Free( size_t size )
```

Bump Allocator

The Bump allocator behaves in a similar way as a general Bump. One of the simplest allocators to implement and better in terms of speed.

Bump Allocator: BumpAlloc() Function

The function signature/prototype:

void *BumpAlloc(size_t size);
Allocates memory of **size** bytes from the Bump allocator.

1. The Allocator first starts with a large chunk of empty memory that was previously allocated from the OS. The allocator has a **current** pointer which points to the **bottom/base** of unallocated memory. Initially, this pointer will point to the starting address. The allocator will also, obviously, need to store the size of the total memory it has allocated from the OS.
2. When Alloc() is called, the allocator checks if enough unallocated memory is present.
3. If enough memory is present, the value of the **current** pointer is incremented by the size of the allocation. The previous value of the **current** pointer (i.e., before the increment) is returned to the caller.

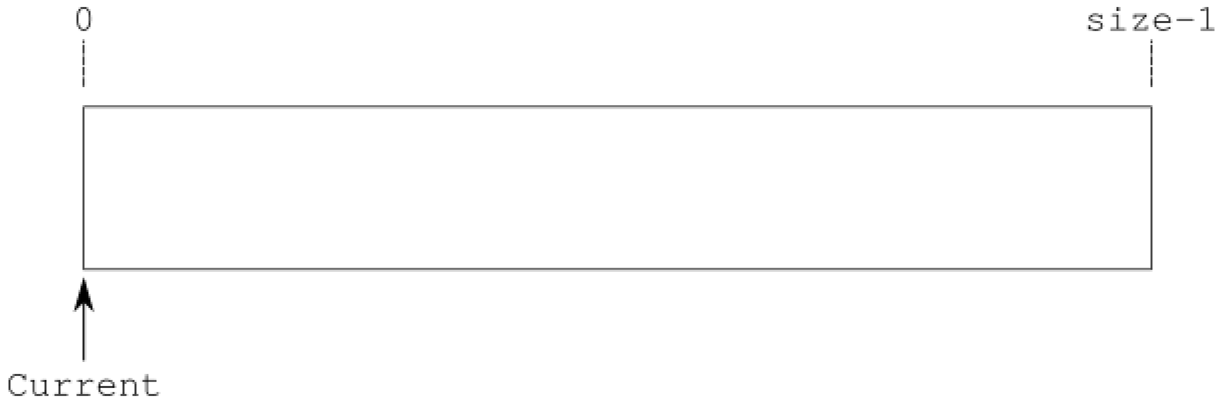


Figure 1: The Bump allocator with **size** bytes allocated from the OS. At the beginning, all **size** bytes of the Bump allocator are free.

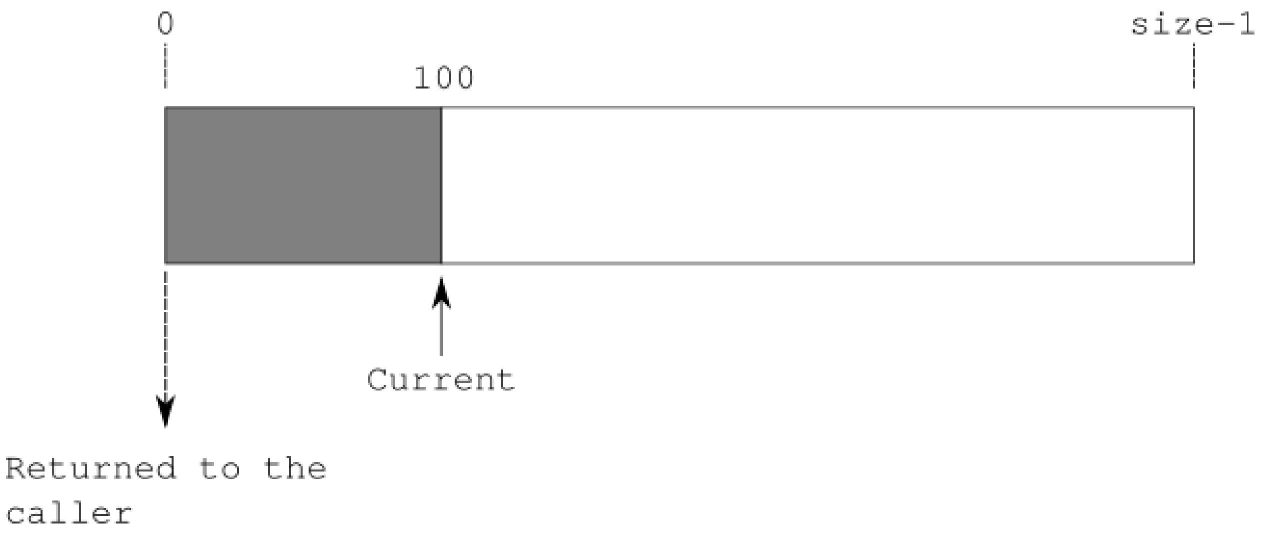


Figure 2: The Bump allocator after the first allocation of 100 bytes (shaded in grey).

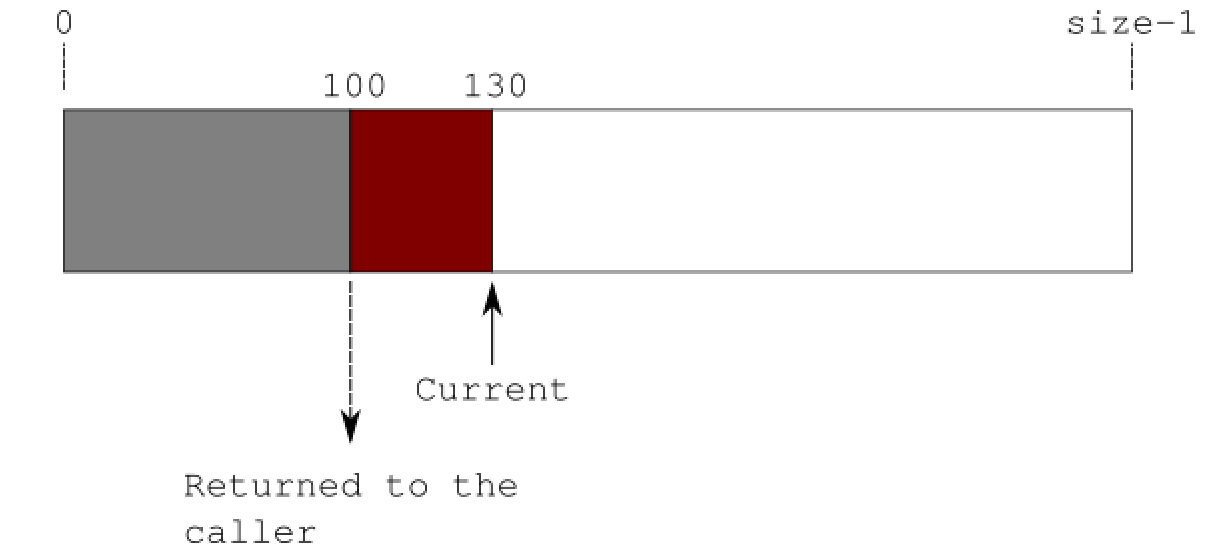


Figure 3: The Bump allocator after a second allocation of 30 bytes (shaded in red).

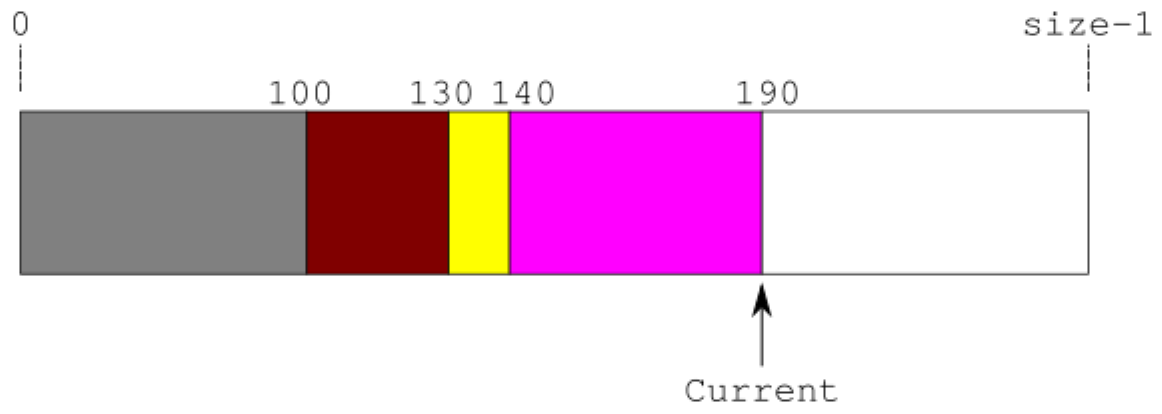


Figure 4: The Bump allocator after a few more allocations. The different colored blocks are allocated by different parts of the program.

Handling Alignment:
Perform bounds check on the **aligned value** of the **current** pointer. Also, return the **aligned** current pointer.

Bump Allocator: Free() Function

The function prototype/signature is:
void BumpFree(size_t size);
Frees/Deallocates **size** bytes from the top of the allocator.

From Figure 4, we can somewhat see that free-ing in a Bump based allocator is not possible, or not as flexible as the CRT **free()** function. Frees in the Bump allocator are simply done by decrementing the **Current** pointer. For this, the caller **must** provide the size of memory to be freed.

In a fashion similar to a stack, only the topmost block can be freed in a Bump allocator (the pink block in Figure 4), i.e, after the call Free(50), the Bump allocator will be as shown in figure 5 below.

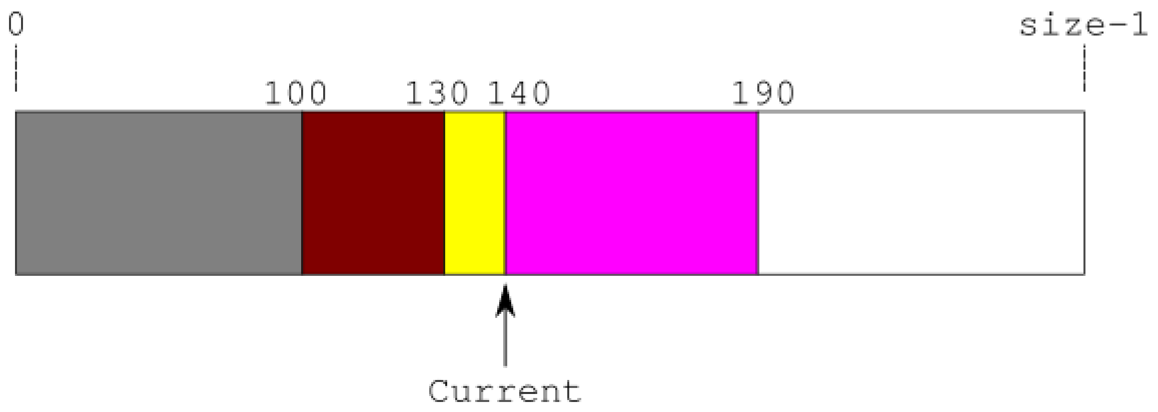


Figure 5: The Bump allocator after freeing the pink block (50 bytes), i.e, after a call to BumpFree(50)

In a simple Bump allocation scheme, the allocator **does not** keep track of the sizes of all the allocations made. Thus, it is up to the user to pass the correct size value to the BumpFree() function, an incorrect size passed to the function can corrupt the memory for allocations made previously.

Because the BumpFree() function is very limited in its use, freeing is not done very often and thus, a Bump allocator is used for purposes where freeing/deallocating the memory is limited or not done at all.

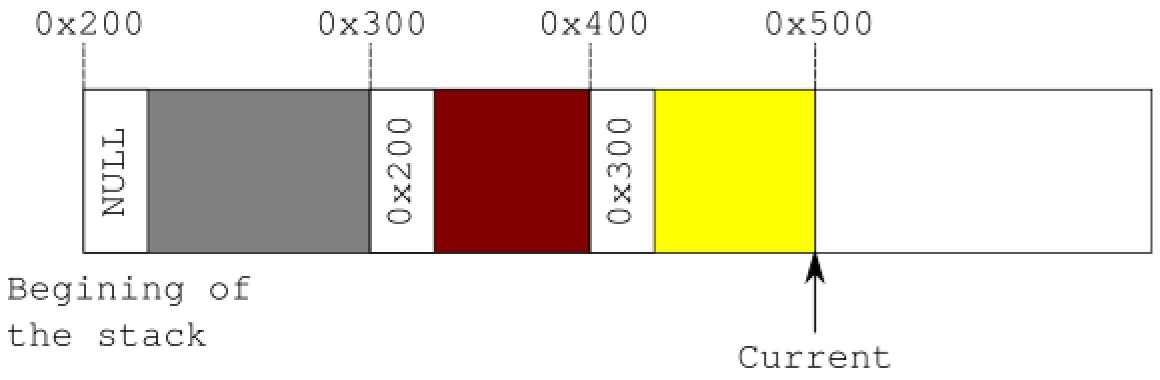
Generally, the Bump memory is freed all at once, i.e, objects are allocated one by one and are freed at the very end all at once. This makes the Bump allocator great for temporary memory in a function.

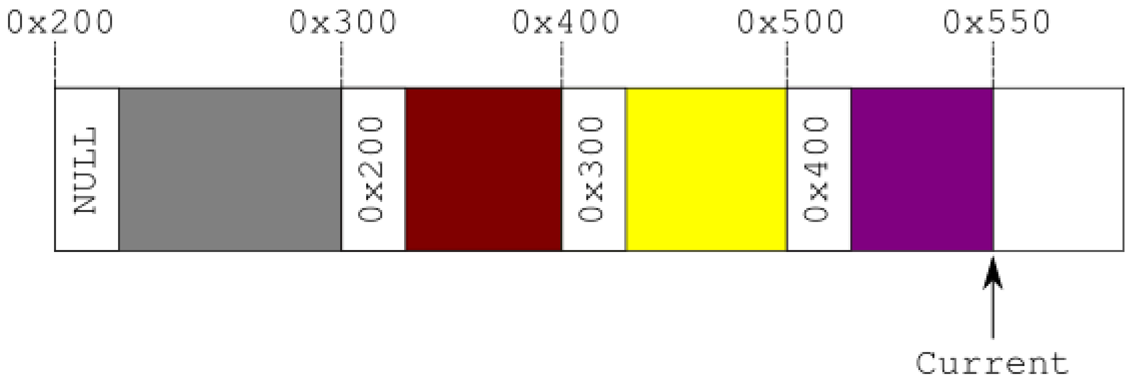
When not enough memory is available:

When the Bump allocator runs out of memory (the other case in step 3 of Alloc() function), we simply return a NULL pointer indicating that the allocator has run out of memory. Realloc-ing and thus, increasing the size of the Bump allocator is not possible, since there is no guarantee that the memory previously held by the allocator will again be in the same address. Therefore, if we realloc in an attempt to increase the size of the allocator, we will most likely invalidate all the pointers of the memory that was previously allocated.

Stack Allocator

The stack allocator is similar to a bump allocator, with the only difference being that the stack allocator will also keep track of allocations, which allows the user to pop off allocations from the top. Thus, the allocator will need to store some additional information as well. The extra information required will be the address of the PREVIOUS value of the current pointer, i.e, we will also need to store the value of the pointer before the allocation. This is shown in the figure below.





Additionally, the stack allocated will also need to keep track of the most recent pointer that is saved, i.e, the value 0x300 in figure (a) and 0x400 in figure (b).

The StackAlloc() Function:
The function signature will be the same as that of bump allocator:
void *StackAlloc(size_t size)
Algorithm:
Input: size => Size of the memory to be allocated.

1. Align the current pointer to a suitable value so that we can store a pointer in that address.
   mem = align_pointer(Stack.current)
2. Store the current TOP value in the address.
   *mem = Stack.top
3. Set the top value to the memory address.
   Stack.top = mem
4. Point to the address immediately after the pointer information.
   mem = mem + sizeof(void *)
5. Allocate size bytes from this pointer, as done in the bump allocator.
   Stack.current = mem + size
6. Return the pointer of the memory.
   return mem

The StackPop() function
The function signature will be:
void StackPop(void)

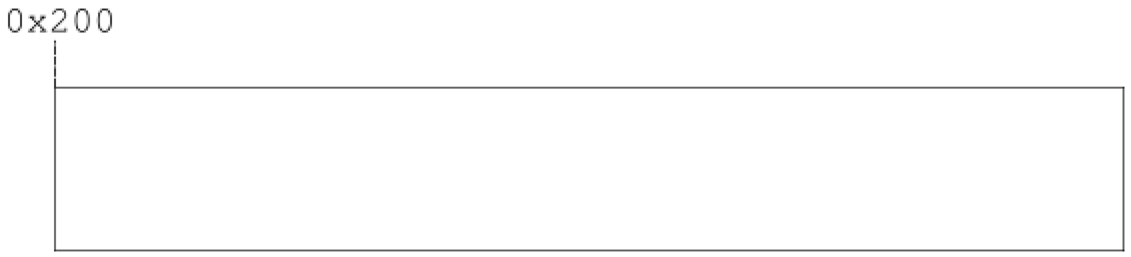
Algorithm:
1. Get the value of the top pointer.
   next = *Stack.top
2. Set the current pointer to the top of the Stack.
   Stack.current = Stack.top
3. Set the top of the stack to the next value.
   Stack.top = next

Free List Allocator

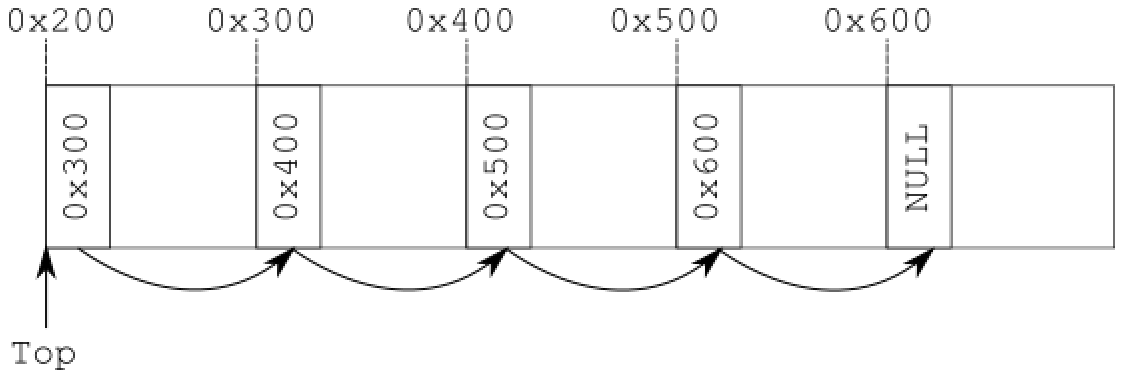
The Free List allocator uses a linked list to store **memory chunks** of a pre-specified size. The free list allocator can be used to allocate memory of a **fixed** size. Also, it allows for freeing of memory blocks, which was not previously possible in a generic way.

How it Works:

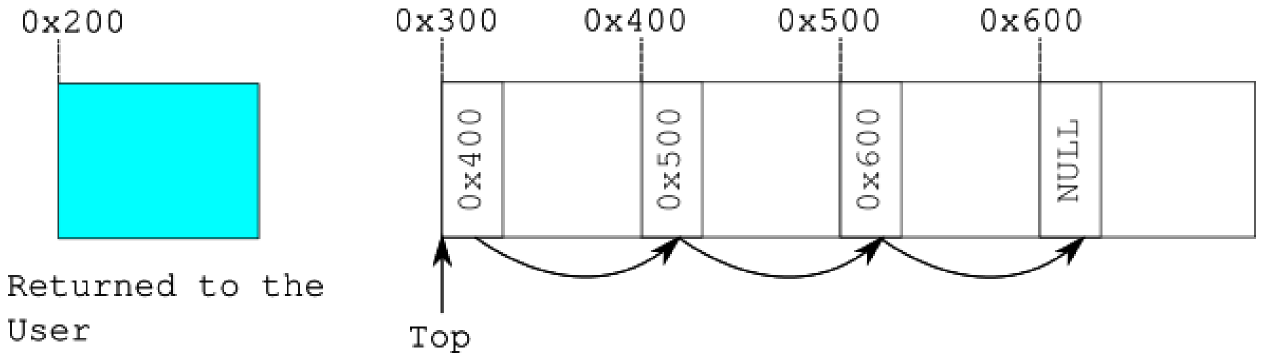
1. Start with an empty page, taken directly from the OS.



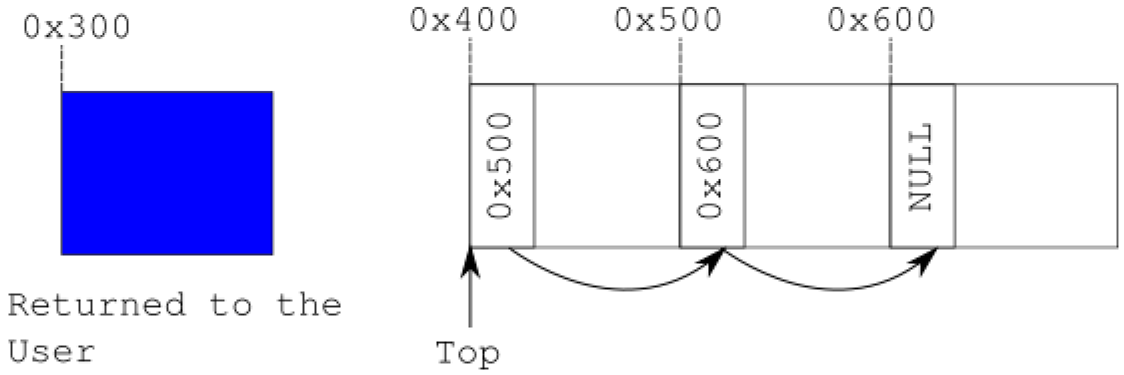
2. Here, we will assume that the **chunk size** will be of 256 (0x100) bytes. Then, we create a linked list of memory chunks.



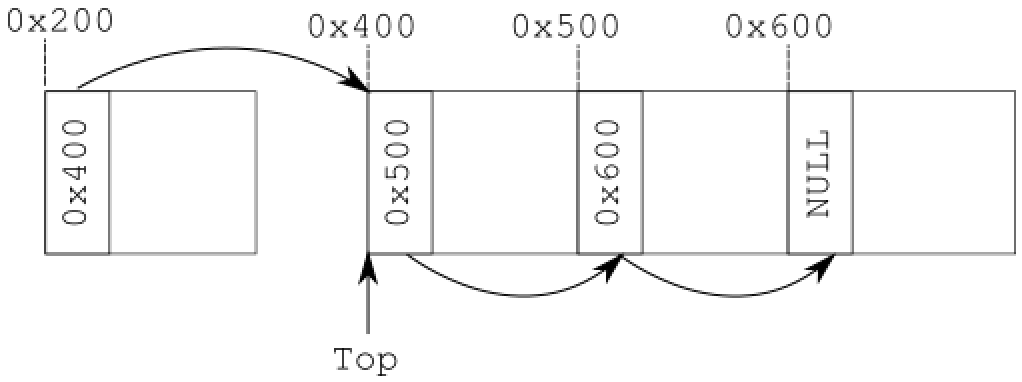
3. To allocate a chunk, we remove the Top node from the linked list and return it to the user. Everything else remains unchanged.



After a second allocation,



4. To free a chunk, we simply add it to the Top (i.e, HEAD) of the linked list. Notice that we need to store the NEXT pointer in the freed chunk as well.



When the free list runs out of chunks to allocate, we can simply allocate a new page from the OS and turn it into a free list. No extra additions are necessary. However, we will need to keep track of all the extra mappings we have created so that we can free it back to the OS when our allocator is destroyed.