Memory API and Address translation mechanisms

What is Memory API ?

* interface provided by OS and runtime to allocate / free / manage memory

Types of memory allocation

Type	Description	Life time	Example
Static	Fixed size, compile time	whole program	int a; (Global variable)
	run time controlled		
Stack	function local variables	Function call	int x; (inside a function), return addresses, etc.
Heap	for dynamic memory allocation	Programmer control	malloc, new, etc.

The malloc() call

You pass it a size asking for some room on the heap, and it either succeeds and gives you back a pointer to the newly allocated space, or fails and returns NULL.

eg:- (1) allocating memory for an integer:

```
int *x = malloc(sizeof(int));
```

(2) allocating memory for an array of doubles.

```
double *arr = malloc(10 * sizeof(double));
```

(3) allocating memory for a string copy

```
char *source = "Hello"
char *copy = malloc(strlen(source) + 1);
```

*Refer to man pages to learn more about malloc().

The free() call : to free heap memory that is no longer in use.

```
int *x = malloc(10 * sizeof(int));
...
...
free(x);
```

Common errors

(1) Not checking if malloc() returns NULL.

If memory allocation fails, malloc() returns NULL. Using the returned pointer without checking can cause crashes.

```
int *ptr = malloc (sizeof(int) * 10);
if ptr == NULL {
    printf ("Memory allocation failed! \n")
    exit(1);
    }
```

(2) using memory after free()

Accessing memory after it has been freed leads to undefined behavior and can cause crashes or data corruption.

```
eg:- int *ptr = malloc (sizeof (int));
     *ptr = 5;
     ....
     free (ptr),
     printf ("%d \n", *ptr);
```

(3) Double free()

freeing the same memory twice causes undefined behavior; often crashing the program.

```
eg:- int *ptr = malloc(sizeof (int));
     free (ptr);
     ....
    free (ptr);
```

(4) Memory leaks

If you don't free() dynamically allocated memory, your program will leak memory, causing increased memory usage.

```
eg:- void func() {
        int *ptr = malloc(sizeof(int) * 100);
     }
```

(5) Incorrect size in malloc()

Allocating less memory than needed causes buffer overflow;

```
eg- char *str = malloc(5);      or
    strcpy(str, "Hello");
```

Forgetting to allocate space for the null terminated strings.

```
eg:- char *str = malloc(strlen("hello"));
     strcpy(str, "hello");
```

Sometimes, allocating more, wastes memory as well!

(6) Forgetting to initialize allocated memory.

When you use malloc(), the allocated memory contains garbage values. Using this uninitialized memory without explicitly initializing it can cause unpredictable behavior.

eg:- `int *arr = malloc (5 * sizeof (int));`

```
for (int i=0 ; i<5; i++) {
    printf("%d", arr[i]);
}
```

Are malloc() and free() system calls? No!

They are library calls that manages space within your virtual address space.

But they are built on top of some system calls → to ask more memory from OS/ release some back to the system.

A few such system calls are

① brk(): takes one argument (an address) and set the program's break (location of the end of the heap) to a specific address.

② sbrk(): moves the program break by a relative increment (+ve or -ve)
eg:- sbrk(n) moves break by n bytes relative to the current break.
which grow the heap linearly

③ mmap(): unlike the above two, mmap() can be used for non-contiguous memory allocations anywhere in virtual address space.
- used extensively for memory-mapped files, shared memory, and large anonymous allocations.

Address translation mechanism

Recall that, inorder to facilitate multi-programming, OS allow multiple programs to reside concurrently in memory.

Virtual address space: every process assumes it has access to a large space of memory from address 0 to MAX (for eg:- let MAX = 16 KB)

We then have the following questions.

* How do we maintain control over which memory locations an application can access?

* How to ensure that application memory access are properly restricted?

* How do we convert a virtual address to physical address?

i.e, How to efficiently flexibly virtualize memory?

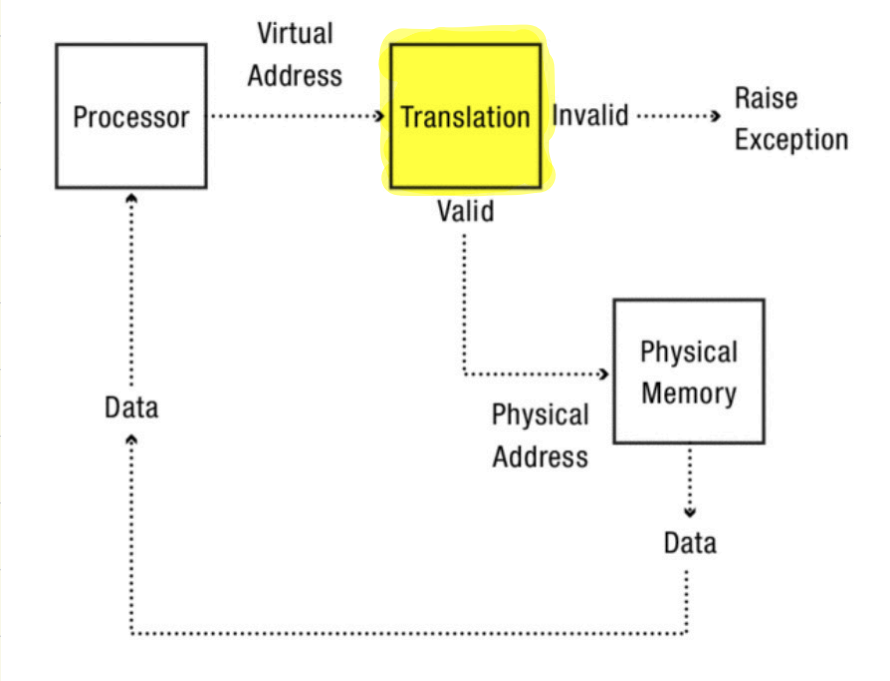
OS with the help of the hardware create this beautiful illusion.

As in the case of scheduling policies, here also we first build up the mechanism based on certain assumptions.

Assumptions

* The user's address space must be placed contiguously in physical memory.
* The size of the address space is not too big; specifically, that it is less than the size of physical memory.
* each address space is exactly the same size.

Goal:



We will explain a basic mechanism to perform above with the help of an example.

eg:- Consider the following code.

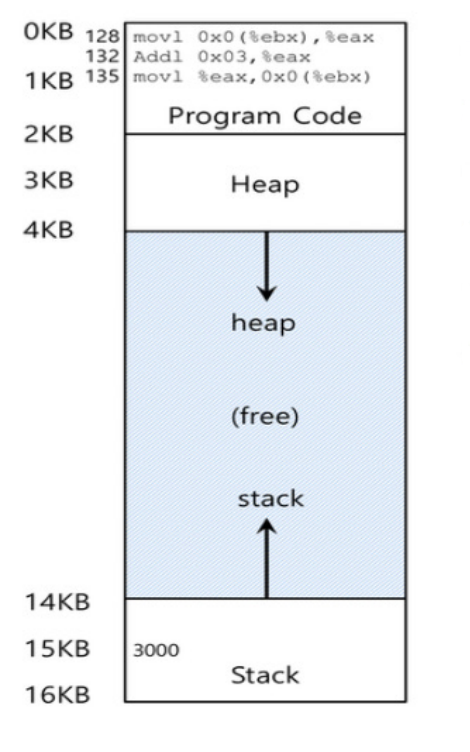
```
void func()
        int x;
        ...
        x = x + 3; // this is the line of code we are interested in
```

* load a value from memory
* increment it by three
* store the value back into memory.

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
```

* load the value at that address into eax register
* add 3 to eax register
* store the value in eax back into memory.



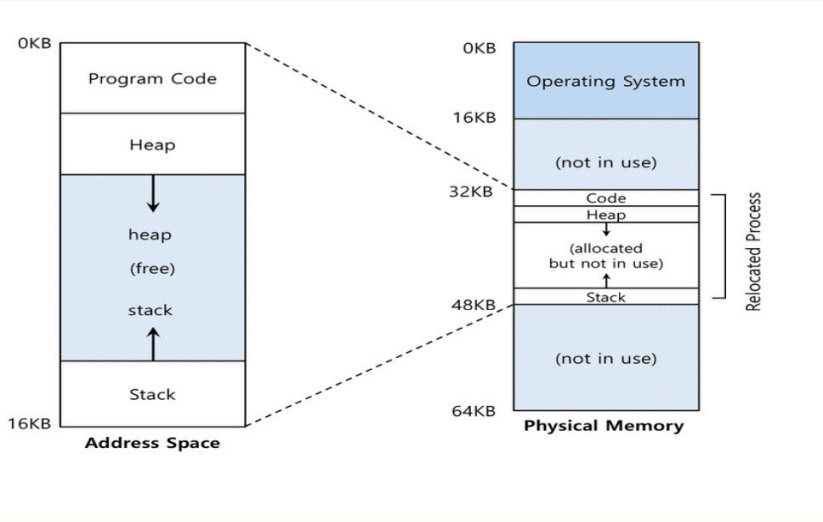
* Fetch instruction at address 128
  Execute this instruction (load from address 15KB)

* Fetch instruction at address 132
  Execute this instruction (no memory reference)

* Fetch the instruction at address 135
  Execute this instruction (store to address 15 KB)

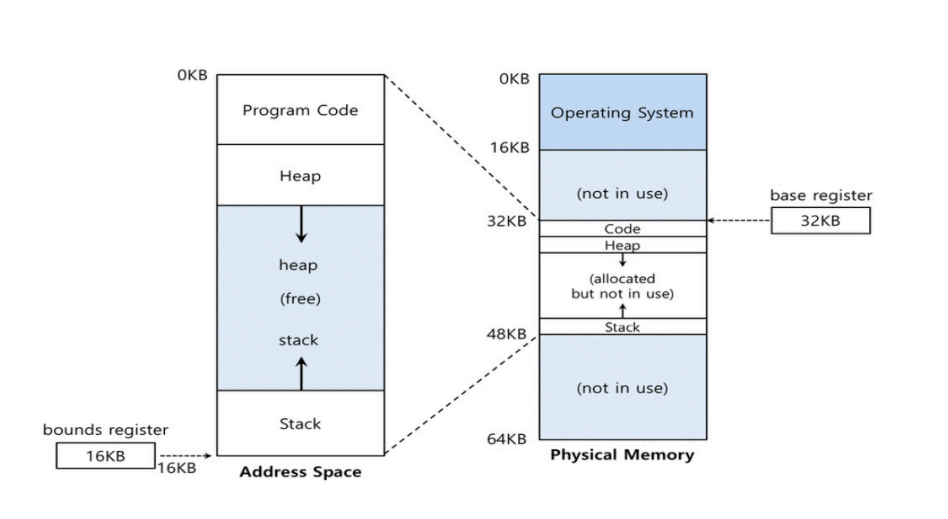
As far as a user is concerned, the address space starts at address 0.

But the OS wants to place the process somewhere else in physical memory, not at address 0.



How to perform this relocation?

Base and Bound Register



when a program starts running, the OS decides where in physical memory a process should be loaded.
(will see later, how this decision is made)

* set the base register a value

⇒ Physical address = Virtual address + base.

* every virtual address must not be greater than bound and negative.

$0 \leq$ Virtual address $<$ bounds.

In the above example,

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

* fetch instruction at address 128

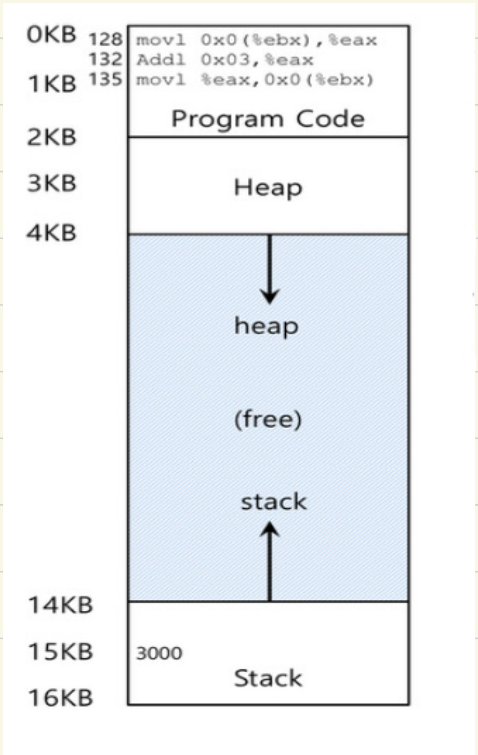
⇒ actual location = 128 + 32KB (base)

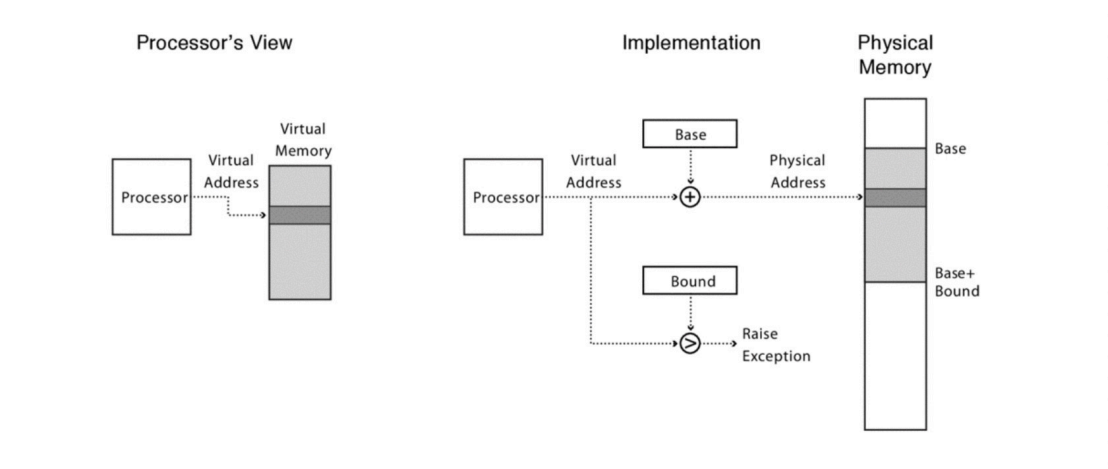
= 128 + 32768 = 32896

* execute this instruction

- load from address 15KB

⇒ actual address = 15KB + 32KB = 47KB





On the above implementation, bound is set to be the size of address space.
In an alternative way, bound could also hold the physical address of the end of address space