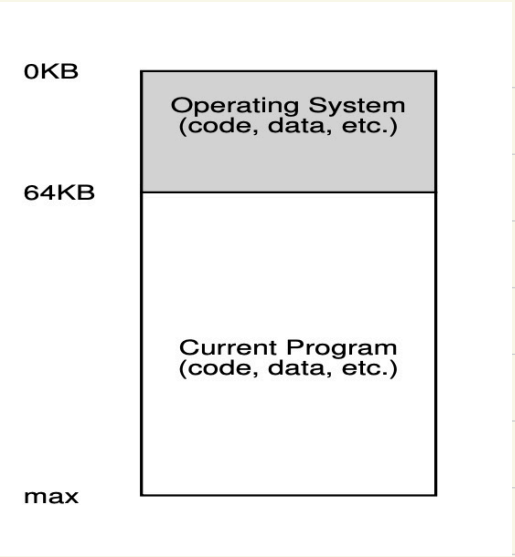
Memory Abstraction



Early systems

* The OS was just a set of routines (more like a library) that sat in a memory.
* There would be one running program (a process) that currently sat at physical memory and used the rest of the memory.
* Users didn't expect much from the OS.

Multiprogramming and Time sharing

* Machines were expensive, people began to share machines more effectively.
* Later, multiple processes were ready to run at a given time.
* The OS would switch b/w them (for example, when one decided to perform an I/O)

One easy approach to implement

* Run one process for a short while, giving it full access to all memory.

* stop it, save all of its state to some kind of disk

* load some other process's state, run it for a while

What is the problem with the above approach? Too slow! why?

- saving and restoring register-level states (the PC, general-purpose registers etc.) is relatively fast, but

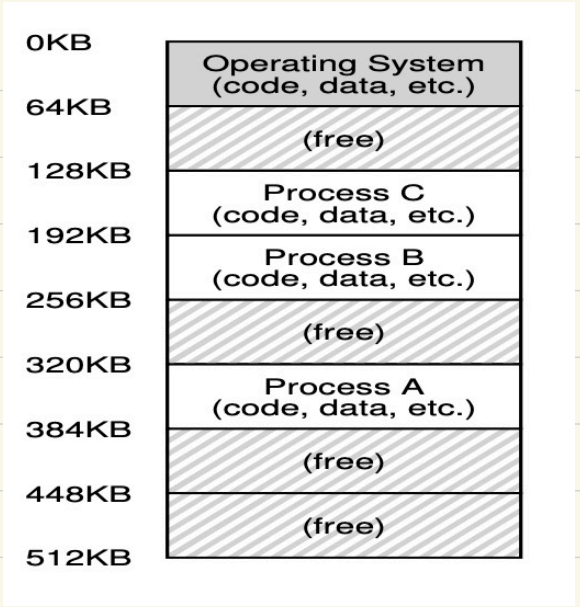
- saving the entire contents of memory to disk is brutally non-performant.

What is the remedy to the above problem?

- leave processes in memory while switching b/w them.

Virtualize memory

* Allow multiple programs to reside concurrently in memory.
* They can be in non-contiguous locations in memory.
* OS doesn't want to reveal this messy picture of memory to its users.



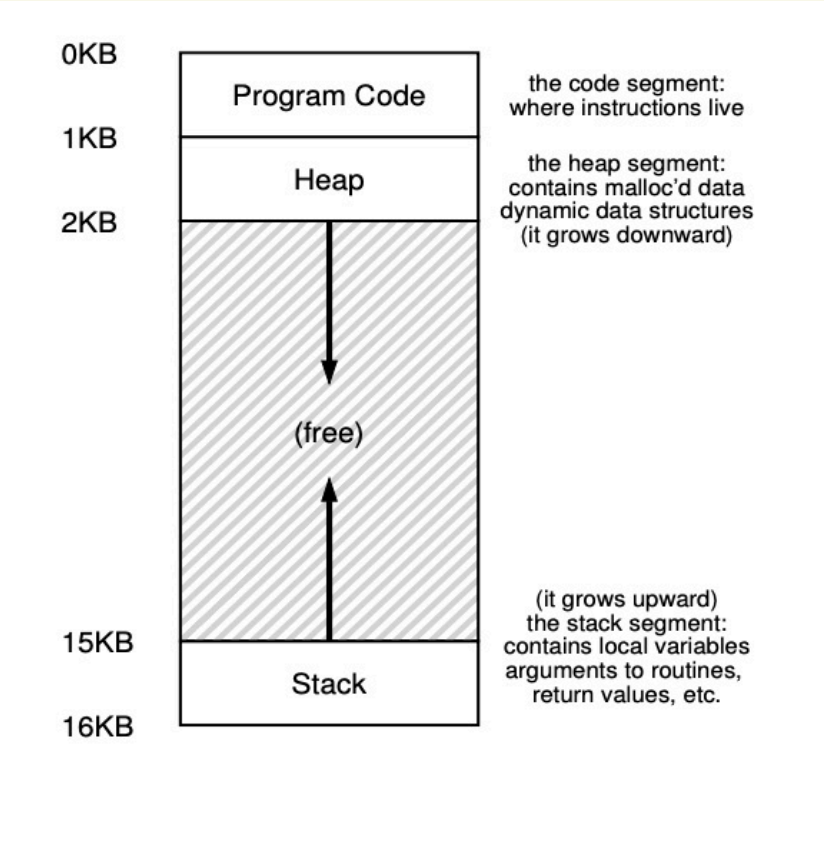
Thus, it creates an easy-to-use abstraction of the physical memory.

This abstraction is called the address space running program's view of memory in the system.

An address space contains all of the memory state of the running program.

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)

* As mentioned in the earlier lectures, the memory image of a process contains
  - code and static data
  - heap (for dynamic allocations)
  - stack (used during function calls)



Note that stack and heap grow during run time.

Remember that this diagram of the memory image of a process is just an abstraction that the OS is providing to the running program.
- the program isn't in memory at physical address 0 through 16 KB. It is loaded at some arbitrary physical addresses.

From virtual address to Actual memory

1. Instruction execution
   The CPU executes a load or store instruction like:

```
load R1, [0X00401234]      // Load from virtual address
                              0X00401234 into register R1
```

2. virtual address issued
   The CPU doesn't directly access physical RAM. Instead it uses the virtual address in the instruction (eg:- 0X00401234)

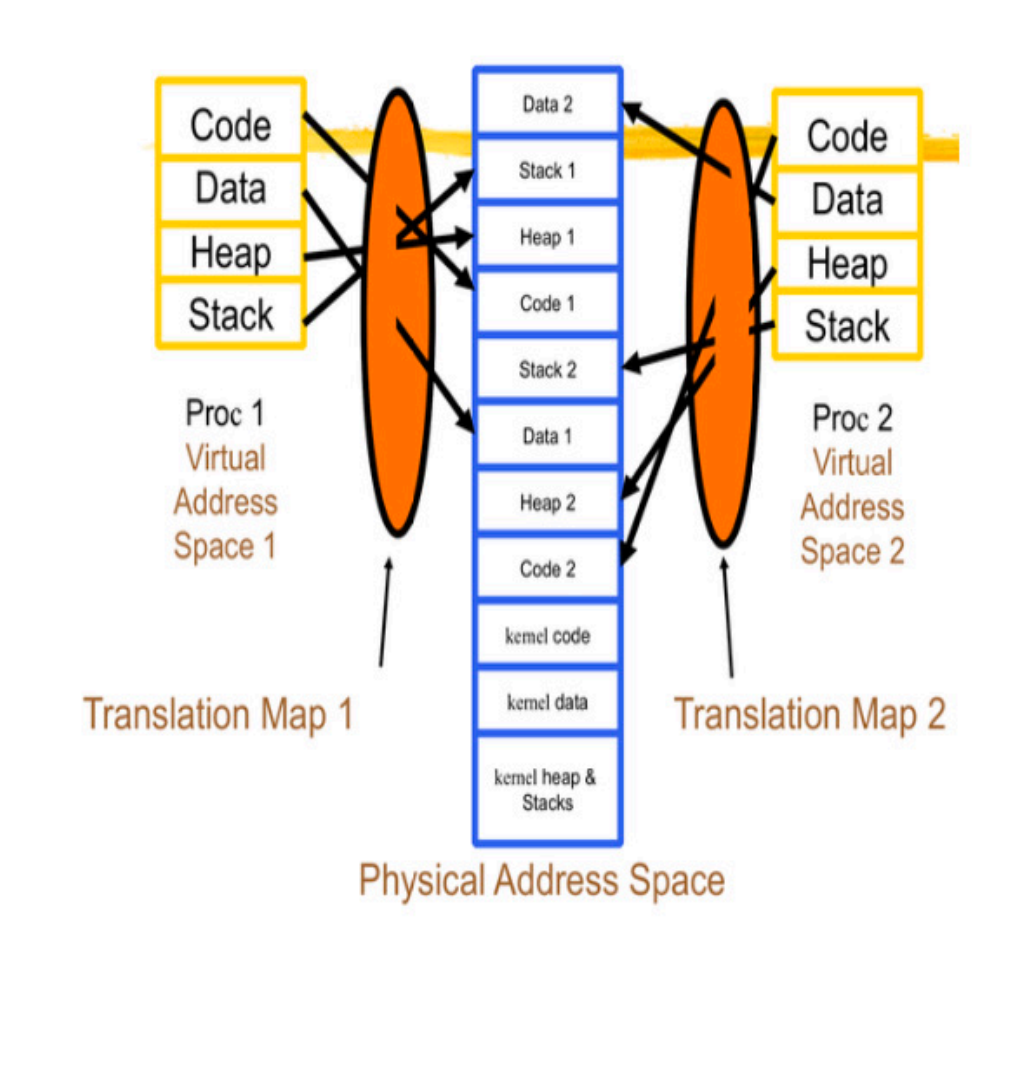
3. MMU translates virtual address 'memory hardware'
   The Memory management Unit (MMU) inside the CPU translates the virtual address to a physical address using page tables.

   (will learn the address translation in detail in upcoming lectures)

   If translation found,
The page table maps virtual address to the corresponding physical address.
   (if translation not found, OS handles "page fault")

4. Physical memory access

Now the CPU can access RAM at the physical address computed by the MMU. Data is read (load) or written (store).



* Note that every address in a running program is virtual.

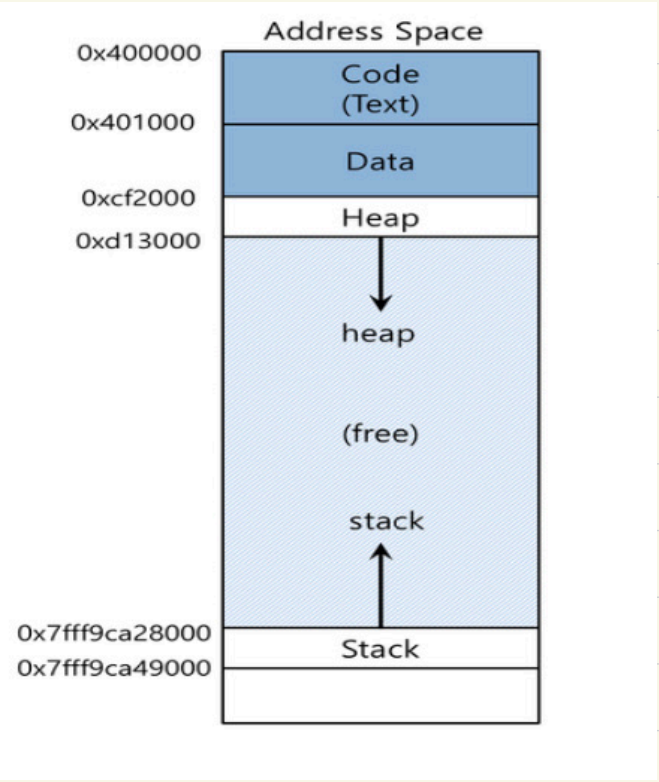
```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]){

    printf("location of code  : %p\n", (void *) main);
    printf("location of heap  : %p\n", (void *) malloc(1));
    int x = 3;
    printf("location of stack : %p\n", (void *) &x);

    return x;
}
```

A simple program that prints out addresses



The output on a 64-bit linux machine

```
location of code  : 0x40057d
location of heap  : 0xcf2010
location of stack : 0x7fff9ca45fcc
```

From the above addresses, we feel like code comes first in the memory, then the heap and then the stack is all the other

end of this large space. But all of this is an illusion, where the true physical locations might be completely different.

Goals of memory virtualization

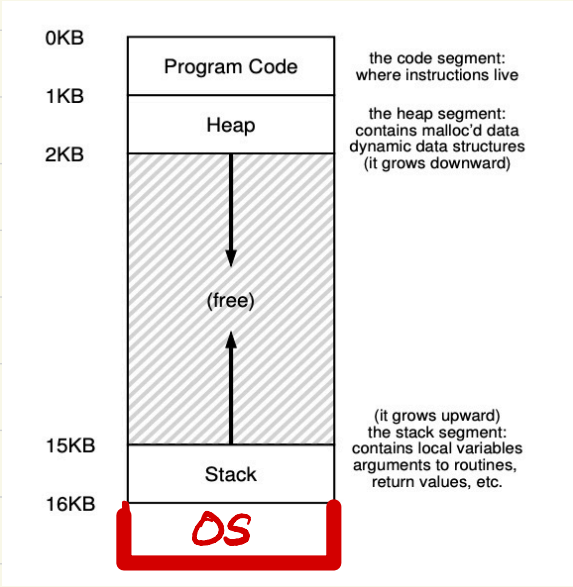
Transparency: user programs should not be aware of the messy details.
OS should implement virtual memory in a way that is invisible to the running program.

Efficiency: minimize overhead and wastage in terms of memory space and access time.
To achieve this, OS will have to rely on hardware support, including features like TLBs (which we will discuss later).

Isolation and protection: a user process should not be able to access anything outside its address space.
OS should protect processes from one another as well as the OS itself from processes.

A final thought: What is the address space of the OS?

* On most modern systems, the OS is not a separate process, but its code and data are mapped into a portion of every user process's address space.
* a process sees OS as a part of its code (eg:- library)
* This kernel space is typically in the upper part of virtual address range.
* This shared kernel mapping is the same in all processes.



Benefits of shared virtual address mapping

* context switching b/w processes is faster.
* System calls can directly access kernel memory using the known virtual address.
* The OS doesn't need to maintain multiple copies or virtual layout of itself.

What is this address

* On 64-bit systems, the OS reserves the upper portion of the virtual

address space for the kernel.

For eg: in Linux X86-64,

0x0000000000000000 - 0x00007FFFFFFFFFFF → user space

0xFFFF800000000000 - 0xFFFFFFFFFFFFFFFF → kernel space

* Note that every user process maps the same kernel space, but it cannot access it in user mode (thanks to hardware-privilege levels!)

* How does OS allocate memory for its own data structures? like PCBs, process queue, etc..

- at boot time the OS uses boot-time allocator to allocate a chunk of physical memory

- after paging is enabled, OS uses more sophisticated memory management techniques! (will see later!)

Why OS cannot use malloc()? malloc() is a part of the C library. When a program calls malloc(), C library sometimes relies on OS system calls (like brk()/sbrk()/mmap()) to allocate memory. So if OS calls malloc() it could happen to call itself and can lead to unfavourable situations.