Memory Virtualization

> Like CPU resources, operating systems provide certain illusions to programs in terms of memory. For example, providing each process the illusion that it has a large personal address space.

Address Spaces

Early Systems

Early systems didn't provide much abstraction. There would just be one running process, and it'd get access to all available memory (besides what was being used by the OS).

Multiprogramming and Time Sharing

Running multiple processes at the same time, or *multiprogramming* complicated things but also allowed for better *utilization.* A naive implementation would be to run a process for awhile, giving it full access to resources, and then save its full state to disk and then switch to another. This is slow, however. A better solution is keeping processes in memory while switching between them. With this sort of solution, *protection* of each processes' memory from each other becomes important.

The Address Space

The *address space* is the basic abstraction of physical memory, the running programs view of the system memory. It contains the program's *code*, a *stack* which tracks location in the function call chain and store local variables, and the *heap* to store dynamically-allocated managed memory. Because the stack and the heap both grow, they are placed at opposite ends of the address space and consume space towards each other (though this is complicated when we introduce threads).

Each program is under the illusion that it's been loaded at the beginning of the address space (at address 0), but obviously this can't be the case. Instead, this is true because the OS *virtualizes memory*, mapping virtual addressed the process is aware of to true physical addresses.

Goals

- *Transparency*: the fact that the OS is virtualizing memory should be invisible to the running process.

- *Efficiency*: the virtualization shouldn't introduce much overhead
- *Protection*: processes should be protected from one another, and the OS itself from the processes