**Chapter 12: A dialogue on memory virtualization**

Beside having virtualization on the CPU, memory is also virtualized.

Every address generated by a user program is a virtual address. Again, the OS is just providing an illusion to each process. With some hardware help, the OS will turn these pretend virtual addresses into real physical addresses.

**Chapter 13: The abstraction: address spaces**

**13.1 Early Systems:**

Early machines did not provide much of an abstraction to users. The physical memory only has code, data, etc. of the OS (starts at physical address 0) and current running program that uses the rest of the memory.

**13.2 Multiprogramming and Time Sharing:**

As machines became expensive, they were shared more effectively. Therefore, the era of **multiprogramming** was born where multiple processes were ready to run at a given time and the OS would switch between them. This increases the effective **utilization** of the CPU and saves a lot of money.

Because of the demand of more of machines, the era of **time sharing** was born. The notion of **interactivity** became important, as many users might be concurrently using a machine. The idea is that one process gets to run with full access for a while, then stop it, save it somewhere and load other process’s state.

Unfortunately, this approach was way too slow because saving the entire contents of memory to disk is brutally non-performant. Thus, what we’d rather do is leave processes in memory while switching between them.

However, this creates an issue as we don’t want a process to read other processes’ memory. Thus, we must take care of the issue of **protection.**

**13.3 The Address Space**

The OS needs to create an **ease to use** abstraction of physical memory. We call this abstraction the address space, and it is the running program’s view of memory in the system.

**The address space** of a process contains all of the memory state of the running program (the code for example). The program uses a **stack** to keep track of where it is in the function call chain as well as to allocate local variables and pass parameters and return values to and from routines. On the other hand, the **heap** is used for dynamically-allocated, user-managed memory, statically-initialized variables, etc.

The program code lives at the top of the address space. Code is static, so we can place at the top and know that it will not need any more spaces as it runs. The heap is at the top while the stack is at the bottom because each of them wishes to be able to grow in opposite direction. This placement is just a convention.

Diagram

Description automatically generated

The program really isn’t in memory at physical addresses 0 through 16KB; rather it is loaded at some arbitrary physical address(es).

When the OS does this, we say the OS is virtualizing memory, because the running program thinks it is loaded into memory at a particular address (say 0) and has a potentially very large address space.

**13.4 Goals**

A major goal of a virtual memory (VM) system is **transparency**. The OS should implement virtual memory in a way that is invisible to the running program. The program should not be aware of the fact that the memory is virtualized. It would rather behaves as if it has its own private memory.

Another goal is **efficiency**. The OS should strive to make the virtualization as efficient as possible, both in terms of time (not making the program runs slowly) and space (not too much memory for structures needed to support virtualization). The OS would need to rely on hardware support.

The third goal is **protection**. The OS has to make sure to protect the processes from one another as well as OS itself. Protection thus enables us to deliver the property of isolation among processes; each process should be running in its own isolated cocoon, safe from the ravages of other faulty or even malicious processes.