
The Abstraction: Address Spaces

In the early days, building computer systems was easy. Why, you ask? Because users didn't expect too much. It is those darned users with their expectations of "ease of use", "high performance", "reliability", and so forth that really have led to all these headaches. Next time you meet one of those computer users, thank them for all the problems they have caused.

9.1 Early Systems

From the perspective of memory, early machines didn't provide much of an abstraction to users. Basically, the physical memory of the machine looked something like Figure 9.1.

The OS was a set of routines (a library, really) that sat in memory (starting at physical address 0 in this example), and there would be one running program (a process) that currently sat in physical memory (starting at physical address 64k in this example) and used the rest of memory. There were few illusions here, and the user didn't expect much from the OS. Life was sure easy for OS developers in those days, let me tell you.

9.2 The Era of Multiprogramming

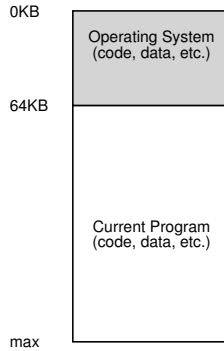


Figure 9.1: Operating Systems: The Early Days

After a time, because machines were expensive, people began to share machines more effectively. The era of **multiprogramming** was upon us, where multiple jobs would be run at once, with the OS deciding which one should run at any given time.

One way to implement multiprogramming would be to run one process for a while, giving it full access to all memory (like the picture above), then stop it, save all of its state to disk (including all of physical memory!), load some other process's state, run it for a while, and thus implement some kind of crude sharing of the machine.

Unfortunately, this approach has a big problem: it is way too slow. While saving and restoring register-level state (e.g., the PC, general-purpose registers, etc.) to the PCB is fast, saving the entire contents of memory to disk is brutally slow. Thus, what we'd rather do is leave processes in memory while switching between them, thus allowing the OS to implement multiprogramming efficiently (as in Figure 9.2).

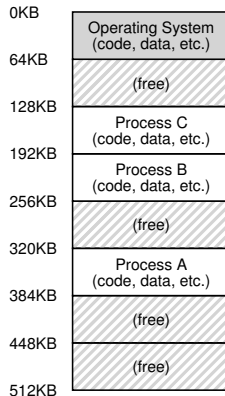


Figure 9.2: Three Processes: Sharing Memory

In the diagram, there are three processes (A, B, and C) and each of them have a small part of the 512-KB physical memory carved out for them. Assuming a single CPU, the OS will choose at any one time to run one of the processes (say *A*), while the others (B and C) would be in the scheduler's ready queue waiting to be run.

The goal of such sharing is to allow for the machine to be used as **efficiently** as possible; for example, when process *A* initiates an I/O and thus moves from the running to the blocked state, the OS can quickly switch to B or C and thus better utilize the CPU.

9.3 The Address Space

However, we have to keep those pesky users in mind, and doing so requires the OS to create an **easy 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. Understanding this fundamental OS abstraction of memory is key to your understanding of how memory is virtualized by the OS.

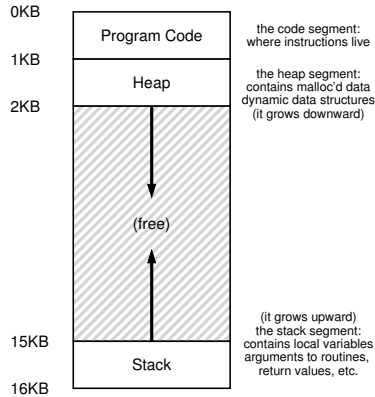


Figure 9.3: An Example Address Space

The address space of a process contains all of the memory state of the running program. For example, the **code** of the program (the instructions) have to live in memory somewhere, and thus they are in the address space. The program, while it is running, 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. Finally, the **heap** is used for dynamically-allocated, user-managed memory, such as that you might receive from a call to `malloc()` in C or `new` in an object-oriented language such as C++ or Java. Of course, there are other things in there too (like statically-initialized variables, and a few other details), but for now let us just assume those three components: code, stack, and heap.

In the example in Figure 9.3, we have a tiny address space

(only 16 KB)¹. The program code lives at the top of the address space (starting at 0 in this example, and is packed into the first 1K of the address space). Code is static (and thus easy to deal with), so we can place it at the top of the address space and know that it won't need any more space as the program runs.

Next, we have the two regions of the address space that may grow (and shrink) while the program runs. Those are the heap (at the top) and the stack (at the bottom). We place them like this because each wishes to be able to grow, and by putting them at opposite ends of the address space, we can allow such growth: they just have to grow in opposite directions. The heap thus starts just after the code (at 1KB) and grows downward (say when a user requests more memory via `malloc()`); the stack starts at 16KB and grows upward (say when a user makes a procedure call).

Of course, when we describe the address space, what we are describing is the **abstraction** that the OS is providing to the running program. The program really isn't in memory at physical addresses 0 through 16KB, rather it is loaded at some arbitrary address(es). Examine processes A, B, and C in Figure 9.2; there you can see how each process is loaded into memory at a different address. And now, hopefully you can see the problem.

THE CRUX: HOW TO VIRTUALIZE MEMORY

How can the OS build this abstraction of a private, potentially large address space for multiple running processes (all sharing memory) on top of a single, physical memory?

When the OS does this, we say the OS is **virtualizing memory**, because the running program thinks it is loaded at an address (say 0) and has a potentially very large address space (say 32-bits); the reality is quite different.

¹We will often use small examples like this because it is a pain to represent a 32-bit address space and the numbers start to become hard to deal with.

When, for example, process A in Figure 9.2 tries to perform a load at address 0 (which we will call a **virtual address**), somehow the OS, in tandem with some hardware support, will have to make sure the load doesn't actually go to physical address 0 but rather to physical address 64KB (where A is loaded into memory).

9.4 Goals

Thus we arrive at the job of the OS in this set of notes: to virtualize memory. The OS will not only virtualize memory, though; it will do so with style. To make sure we do so, we need some goals to guide us; here are a few obvious ones:

- **Transparency:** the OS should do this in a way that is **transparent** to the running program. The program behaves as if it has its own private memory. The OS (with hardware support) does all the work to multiplex memory among many different jobs.
- **Efficiency:** the OS should strive to make the virtualization as **efficient** as possible. As we will see, the OS will have to rely on hardware support for this, including hardware features such as TLBs (which we will learn about in due course).
- **Protection:** finally, the OS should make sure to **protect** both processes from one another and the OS itself from processes. When one process performs a load, a store, or an instruction fetch, it should not be able to access or affect in any way the memory contents of any other process or the OS itself (that is, anything *outside* its address space).

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.

PRINCIPLE OF ISOLATION

Isolation is a key principle in building reliable systems. If two entities are properly isolated from one another, this implies that one can fail without affecting the other. Operating systems strive to isolate processes from each other and in this way prevent one from harming the other. By using memory isolation, the OS further ensures that running programs cannot affect the operation of the underlying OS. Some modern OS's take isolation even further, by walling off pieces of the OS from other pieces of the OS. Such **microkernels** thus may provide greater reliability than typical monolithic kernel designs.