【OS】Day16

【Ch13】The Abstraction: Address Spaces

13.2 Multiprogramming and Time Sharing

In early days, in order for multiple processes to run on memories. The contents of a process on memories are copied onto the disk and restored back to the memory when the process regains control of the CPU. This is brutally non-performant.

Thus, as time sharing became more popular, allowing multiple programs to reside concurrently in memory makes protection an important issues. We don't want a process to be able to run/write any other process' data.

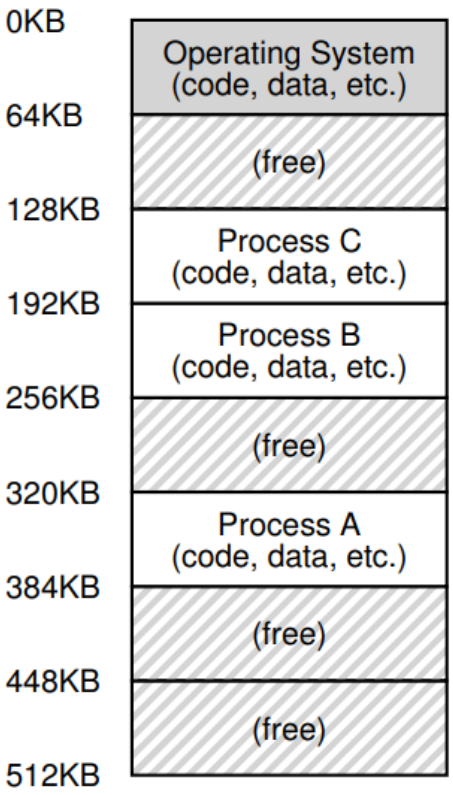

Figure 13.2: **Three Processes: Sharing Memory**

13.3 The Address Space

We call the abstraction of physical memory 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.

- 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.
- Finally, the heap is used for dynamically-allocated, user-managed memory, such as that you might receive from a call to `malloc()` in C.
- Of course, there are other things in there too(statically-initialized variables)

For now, we will assume there are three components: code, stack, and heap.

In the following example, we have a tiny address space.

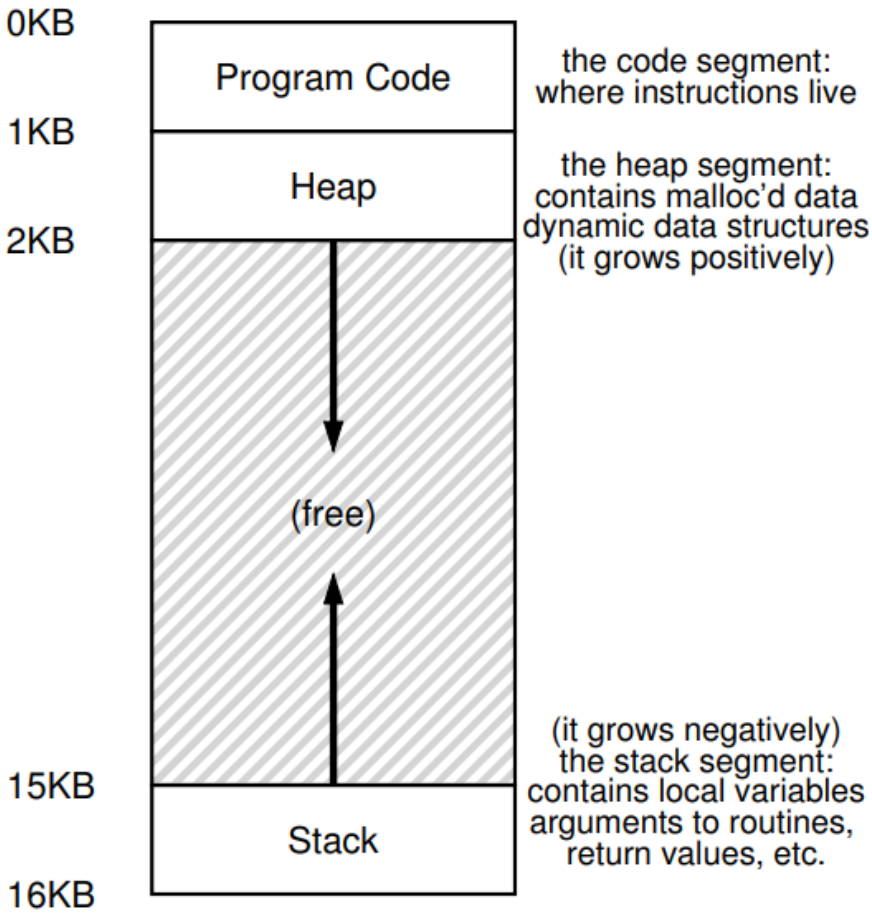

Figure 13.3: **An Example Address Space**

The program code lives at the top of the address space(starting at 0 in this example). Code is static, 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.

However, this placement of a stack and heap is just a convention; we could arrange the address space in a different way if we'd like.

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 into memory at a particular address(say 0) and has a potentially very large address space.

13.4 Goals

One major goal of a virtual memory system is transparency.

The OS should implement virtual memory in a way that is invisible to the running program. Thus, the program shouldn't be aware of the fact that memory is virtualized; rather, the program behaves as if it has its own private physical memory.

Another goal is efficiency. The OS should strive to make the virtualization as efficient as possible, both in terms of time and space.

Finally, a third VM goal is protection. The OS should make sure to protect processes from one another as well as 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.