Address Spaces

Hello, and Welcome to this lesson on Address Spaces.

The term "address space" is awfully abstract, but by the end of this lesson you will have gained a good grasp on how people in the community use it.

We will first look at a simplified picture of how the operating system and one or more programs are laid out in the machine's physical memory.

In modern operating systems, the process is not aware of physical memory. Rather, it lives in what we call "virtual" memory, which is a convenient abstraction of memory that hides the necessary multiplexing that the hardware and the operating system have to do when multiple processes share the available physical memory.

We will have a closer look at how this process memory looks.

On the way we will have gained a good understanding of address spaces in general and of the separation of physical and virtual address spaces.

Let's begin.

The Physical Memory Space

Let's start with the by now well-known system with a simple operating system and several programs sharing the rest of the memory.

The physical memory of the system starts at address HEX zero, and, assuming that the system has 4MB of memory, ends at HEX 6 F's.

Let's further assume that Job 1's memory starts at about 320 odd thousand bytes into the memory and ends at about 720 odd thousand bytes. So the start address is HEX 4F447, and the end address is HEX B0EC4.

Let's have a closer look at this job.

If you were the programmer having to write the code for this job, you would be rather puzzled.

How can somebody write code when the address boundaries are arbitrary, and – to make matters worse – change every time we run the program? There has to be a better way to represent the memory of this job.

A convenient way would be to have the address space of the program, note the use of the term "address space" here, start at address zero, and end at some convenient address as well, say 1MB, that HEX 5 Fs.

This is convenient way for the programmer to work. The program in job 1, which we now can call a "process", refers to addresses that are part of the process's "address space", which is a virtual space. We say that the addresses that the program refers are "logical" addresses, also called "virtual" addresses.

Of course this immediately becomes a problem when we have other jobs in the system, which have their own address space that start at address zero.

Of course the conflict happens at logical or virtual address level, and we will soon see how seemingly conflicting logical memory addresses get resolved to unique physical addresses by the hardware.

The Logical/Virtual Memory Space

Let's forget for now about all these address resolution complications, and let's focus on the address space of a single process instead.

The address space of the process starts, as we mentioned earlier, at address zero and it goes to the end of the processes address space. Let's give this process a 1MB address space, and the end address becomes HEX 5 Fs.

We call this the address space of the process. Sometimes it is called the logical address space or virtual address space of the process as well. Addresses in this address space are called "logical addresses" or "virtual addresses". We will see later that systems can support arbitrarily large address spaces, independently of the size of the available physical memory. For example, today's 64-bit processors support address spaces of size up to 2^64 byte. That's a lot of bytes.

How is this address space used? As we will see, quite sparsely. An example memory organization of a process could look similar to the following.

Near the beginning of the address space is stored the executable code of the program.

In a separate portion of the address space, here at the opposite end, we have the stack of the process. The stack is where the function arguments, return addresses, and local variables are stored. The stack therefore grows and shrinks during the execution of the program.

The third common memory area is the heap, where dynamically allocated data is stored. In C, this is where the allocation function **malloc** gets the memory from. In C++, the operator **new** retrieves memory from the heap. As memory is requested, the heap may grow.

We observe that much of the address space is free space. This is common in real systems. 32-bit machines, for example, support 4GB address spaces, and programs rarely make use of any significant fraction of this memory. The hardware can leverage this to multiplex multiple address spaces onto comparatively tiny amounts of physical memory.

Address Spaces

We have come to the conclusion of this lesson on address spaces.

We learned to distinguish physical memory and the physical address space of the machine, from the virtual or logical address space of a process.

We have also looked at the memory layout of such a process.

I genuinely hope that you enjoyed this lesson on address spaces and that it helped you get a clear idea of the distinction of physical and virtual address spaces. This will help you as we explore how the operating system, in conjunction with the hardware, provides users with very convenient ways to address, allocate, and share memory.

Thank you for watching.