Intro to Virtual Memory

Andrew Quinn

Learning Objectives:

1. What goals do we have for virtual memory?
2. How did historical virtual memory designs fall short?

Announcements:

1. Project 2 is out!

Why virtualize memory?

We have seen how an operating system gives each application running on a system "limited direct execution" (LDE). One of the many goals of LDE is to provide an application with the illusion that it is the only process executing on the machine; a *BIG* part of this illusion is that each process on the system has its own address space.

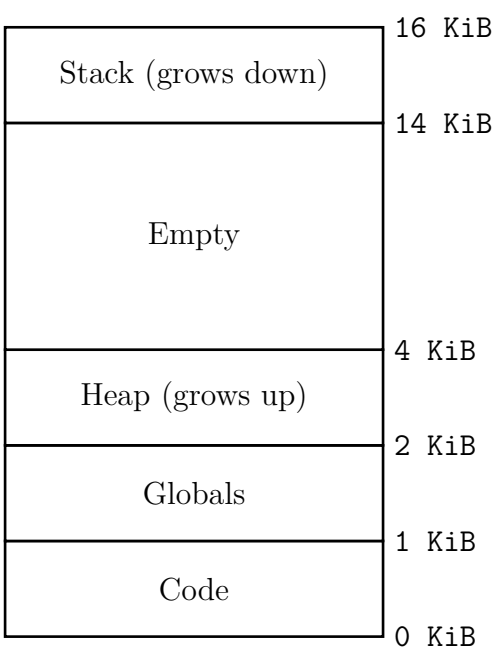
We often draw fig. 1a as our depiction of an application's address space. The address space consists of program code, global variables, a stack, and a heap. We've simplified the drawing a bit here and made a few assumptions, but it's the picture that we'll go with for this unit.

But, as we know from this class, the operating system *actually* has memory that looks like fig. 1b. Virtual memory is what allows the application to "see" fig. 1a, even though the actual system *has* fig. 1b.

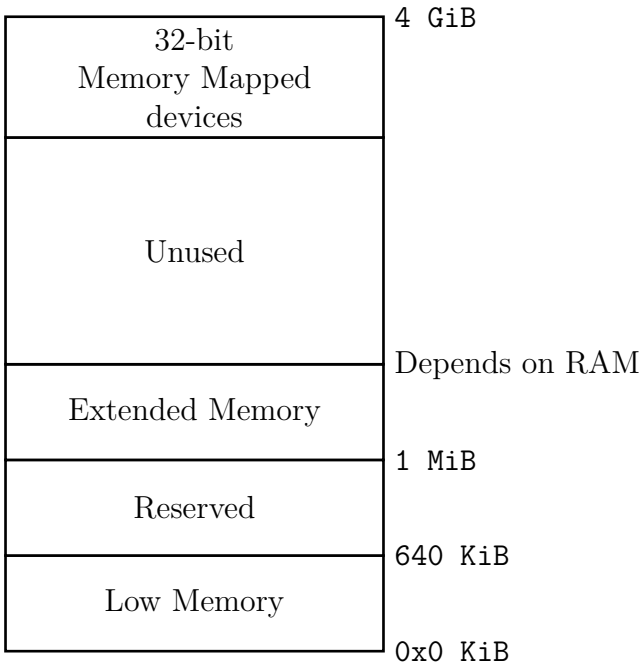
Virtual memory involves both systems software and computer architecture—it is one of the earliest examples of hardware-software co-design in computer systems. This class will focus on the operating system's role in virtual memory, but we will necessarily talk some about how the hardware/architecture plays a role in these systems. You will probably have heard some of this material before in prior computer architecture classes. That's OK, virtual memory is one of those very important topics, so it's good that you'll get more than one chance to learn it.

Let's talk about the goals of a virtual memory system:

1. Protection: Virtual memory aims to ensure that each system and the operating system itself have independent memories or address spaces. This protection may not only apply across applications, but also within x3an application. So called "fine-grained" protections would encompas features such as enforcing protections on parts of a program's memory, such as ensuring that code is not writable, or that the stack is not executable.
2. Efficiency: Virtual memory aims to provide virtual memory with as little runtime overhead as possible. This is the overall aim of LDE. Spacial efficiency is also important for virtual memory: the system aims to provide virtual memory using as little extra memory space as possible.
3. Transparency: Virtual memory aims for *all userspace programs* to be unaware that virtualization takes place. The virtual memory systems in the operating system do not rely upon compilers, loaders, or other "untrusted" software for this feature.



(a) An applicaitons address space.



(b) The memory layout of an 80x86 machine.

4. Expansiveness: Virtual memory aims to allow systems to address more memory than the system actually has. In the early days, expansiveness involved storing some process memory on disk, which is most of what we'll talk about in class.

EARLY DESIGNS

One of the earliest virtual memory designs actually copied process's memory to-and-from disk on every context switch! This design protected applications and was transparent. But, it is not efficient nor expansive. So…

BASE-AND-BOUNDS

Base and bounds uses a linear shift of a process's memory. Kernel virtual memory in PintOS actually acts *very* similar to base-and-bounds, as does memory in certain programming languages (e.g., WebAssembly's "linear memory" is essentially a base-and-bounds implementation in userspace). Here's how it works:

- Applications get a contiguous virtual memory space that starts at 0 and ends at an application specifiable `bounds`. For example, in fig. 1a, the `bounds` would be 16KiB. Note: the virtual memory space always starts at 0 in base-and-bounds systems.
- The system assigns each process a contiguous region in physical memory starting at the process's `base`.
- To dereference a virtual memory address, `a`, the system then first checks if `a` is less than `bounds`, and then calculates the physical address as `a` + `base`.

In addition to the hardware for LDE, the hardware/architecture in base-and-bounds provides…

1. Registers that store an application's base-and-bounds. This is part of a hardware unit called an *Memory management Unit* (MMU).
2. Privileged instructions to modify the base-and-bounds.
3. Ability to raise exceptions when a bad access occurs (e.g., when a virtual address is larger than `bounds`).
4. Ability to translate memory addresses during execution.

And, the operating system provides…

1. Ability to update base-and-bounds on context switch.
2. Ability to handle translation exceptions.
3. Memory management:
    1. Ability to allocate memory to a new process.
    2. Ability to free memory on process termination.

People complain about base-and-bounds frequently, but it actually does some things very well:

- Fully transparent.
- Achieves protection.
- Efficient runtime.
- Very simple OS and architecture design.

But, what does base-and-bounds not do well?

- Not expansive.
- No fine-grained protection.
- User's AS must be sized for the maximum possible use (How would you implement virtual memory growth?).
- Leads to *fragmentation*:
    - Internal Fragmentation: space allocated in a unit is not all used.
    - External Fragmentation: space between allocation units cannot all be used.

See course lecture video for an example.

SEGMENTATION

So, base-and-bounds seems bad. What if we generalize it slightly? Rather than having a single base-and-bounds region for the whole address space, let's instead have a group of base-and-bounds regions for the process. We'll call each of them a *segment*. The exact number of segments depends on the particular hardware, but let's simplify by assuming that we have a code segment, stack segment, and data segment. Here's how it works:

- Applications get a set of segments (e.g., code, data, and stack). Each segment is a contiguous virtual memory space that starts at 0 and ends at an application specifiable segment `bound`.

- The system assigns each segment to a contiguous region in physical memory starting at the segment's `base`.
- To dereference a virtual memory address, `a`, the system then first checks if `a` is less than its segment's `bounds`, and then calculates the physical address as `a` + `segment_base`. There are three main ways to track the segment for each memory address:
  - Implicitly as a part of each instruction.
  - Using special registers in each memory access.
  - Encode the segment into the high-order bits of the address (e.g., the first two bits).
- System can provide per-segment memory protections.
- System can provide per-segment growth.
- System can even provide segment sharing across applications.

In addition to the hardware for LDE, the hardware/architecture for segmentation needs…

1. Registers that store base-and-bounds for each segment (MMU).
2. Privileged instructions to modify segment registers.
3. Ability to raise exceptions when a bad access occurs—but, now this needs to check for specific permissions (read or write).
4. Ability to translate memory addresses during execution.

And, the operating system provides…

1. Ability to update segments on context switch.
2. Ability to handle translation exceptions.
3. Free memory management:

    1. Ability to allocate memory to a new process.
    2. Ability to free memory on process termination.
    3. Ability to grow segments

Segmentation gets a lot of things "right":

- Fully transparent.
- Achieves protection.
- Achieves fine-grained protection.
- Efficient runtime.
- Simple OS and architecture design.
- Solves internal fragmentation.

But, segmentation still suffers from external fragmentation. Indeed, the ability to grow a segment surely will make external fragmentation *worse* that in base-and-bounds. Additionally, we have not described any mechanism to provide expansiveness. It seems possible to do it, but also would be very complex and challenging to build.

We'll talk about paging and how it solves these challenges next. One note, though: fragmentation is a fundamental challenge that all allocators face. So, while paging will basically eliminate external fragmentation: (1) we'll see internal fragmentation arise and (2) library-level allocators (e.g., `malloc`) will face the same problems that we've discussed.