Memory Virtualization

> Like CPU resources, operating systems provide certain illusions to programs in terms of memory. For example, providing each process the illusion that it has a large personal address space.

Address Spaces

Early Systems

Early systems didn't provide much abstraction. There would just be one running process, and it'd get access to all available memory (besides what was being used by the OS).

Multiprogramming and Time Sharing

Running multiple processes at the same time, or *multiprogramming* complicated things but also allowed for better *utilization*. A naive implementation would be to run a process for awhile, giving it full access to resources, and then save its full state to disk and then switch to another. This is slow, however. A better solution is keeping processes in memory while switching between them. With this sort of solution, *protection* of each processes' memory from each other becomes important.

The Address Space

The *address space* is the basic abstraction of physical memory, the running programs view of the system memory. It contains the program's *code*, a *stack* which tracks location in the function call chain and store local variables, and the *heap* to store dynamically-allocated managed memory. Because the stack and the heap both grow, they are placed at opposite ends of the address space and consume space towards each other (though this is complicated when we introduce threads).

Each program is under the illusion that it's been loaded at the beginning of the address space (at address 0), but obviously this can't be the case. Rather, this seems true because the OS *virtualizes memory*, mapping virtual addressed the process is aware of to true physical addresses.

Goals

- *Transparency*: the fact that the OS is virtualizing memory should be invisible to the running process

- *Efficiency*: the virtualization shouldn't introduce much overhead
- *Protection*: processes should be protected from one another, and the OS itself from the processes

Interlude: Unix Memory API

A fair bit is omitted about `malloc`, `free`, etc.

Stack memory is managed implicitly by the compiler. For example, a local variable being initialize involves an implicit allocation on the stack by the compiler. *Heap* allocations are used when long-lived memory is required, and are handled explicitly by the programmer.

Mechanism: Address Translation

Memory virtualization pursues a similar strategy to CPU virtualization, using hardware support. The general approach is *hardware-based address translation*, where hardware transforms each *virtual* address into a *physical* one. Alongside this mechanism, the OS helps set up hardware and manages memory. The goal is to create a illusion that each program has its own private memory.

We begin with several simplifying assumptions:

- The address space must be placed contiguously in physical memory
- The size of the address space is smaller than physical memory
- Each address space is the same size

From the point of view of a running process, the address space starts at address 0 and grows to at most 16 KB. For practical reasons, we want to have this process be actually be running somewhere else, and then transparently (to the process) relocate the process.

Dynamic (Hardware-based) Relocation

aka base and bounds

Each CPU has two registers, `base` and `bounds` that indicate the start and end of the address space. While the program runs each address is increased by the value of the base register. To provide protection, the *memory management unit (MMU)* checks that a memory access is within bounds of the process. "Dynamic" from the name refers to the fact that address relocation can happen dynamically at runtime (by changing `base` or `bounds`).

Hardware Support: A Summary

Thus far, we've introduced to

- *Privileged and user modes*, where the CPU provides a limited permission scope to user programs
- *Base and bounds registers*, which are a simple mechanism for address translation
- *Exceptions*, which allow the CPU to preempt a user program and run an *exception handler* when it executes an illegal instruction (like trying to access illegal memory)

Operating System Issues

things the OS has to do

- When a new process is created, the OS needs to find memory for its address space. Under our simplifying assumptions (all address spaces are the same size, etc.), this is easy, but in realistic systems it involves some sort of *free list*.
- When a process is terminated, it needs to reclaim its memory.
- State must be managed during context switches. Since there is only one *base* and *bounds* register each, they must be saved and restored into something like a *process control block* for address translation to work correctly.
- The OS must provide *exception handlers*, prepared at boot time, so it knows what to do when exceptions occur.

Segmentation

Base and bounds is a fairly simple implementation that meets some of the important goals of memory virtualization (transparency, efficiency, protection). However, there are also downsides. In particular, there can be substantial *internal fragmentation*, when lots of space between the stack and heap isn't used. On a 32 bit system, each program would have an address space of nearly 4 GB, but would typically use only a few MB. This motivates a generalization called *segmentation*.

Segmentation: Generalized Base/Bounds

Because *code*, *stack*, and *heap* are logically separate segments, by placing each separately into different parts of physical memory we can only allocate space to used memory. To support this, we have three pairs of *base-bounds* pairs – one for each segment. We can tell which segment an address belongs to explicitly

by checking bits in the address (ex. the top two bits, and the rest storing the offset) or implicitly based on usage. Also, since the stack grows in a different direction from the heap, we require an additional hardware register tracking which direction a segment grows.

Support for Sharing

In the interest of efficiency, the OS can also support memory sharing. For example, the OS might be able to save memory by having several processes read from the same memory. Additional *protection bits* are used to implement this, indicating the read/write permissions a program has for each segment. When a user program violates the permissions specified by the protection bits, hardware raises an exception.

Fine-grained vs. Coarse-grained Segmentation

Just as segmentation split up segments more finely than base-bounds pairs, various operating systems like Multics have attempted even more fine-grained segmentation. To manage this, they would maintain some sort of *segment table*. The goal was that the OS could use the additional information to better organize memory.

OS Support

Segmentation pretty effectively solves the internal fragmentation problem of base-bounds, but introduces new ones. As processes start, memory becomes full of little holes, making it progressively harder to allocate new segments (known as *external fragmentation*). A naive approach is to periodically stop and relocate all processes, compacting memory. This is expensive though, so practical algorithms like *best-fit* and the *buddy algorithm* try to manage free space as well as possible. It's difficult to come up with a good general solution though.

Free Space Management

The challenge is managing free space is figuring out how to avoid fragmentation as variable-sized chunks of memory get allocated. We want to avoid situations when we're unable to satisfy a memory request even though there's enough total space, just because there isn't enough contiguous space in memory.

There are several possible interfaces for user-level memory allocators. The familiar one is `malloc(size)` and `free()`, and the memory managed by the library is known as the *heap*. Under the hood, there's some sort of *free list*

to track all the chunks of memory that are unallocated within the region of managed memory.

As we consider ways to minimize *external fragmentation*, we'll assume for simplicity that *compaction* isn't possible, and that the managed memory reason is a contiguous section of memory.

Low-level Mechanisms

a few general techniques that are used in most allocators

Splitting and Coalescing

Suppose we have the following memory layout of 30 bytes:

|| 10 free || 10 allocated || 10 free ||

We won't be able to satisfy a request for anything more than 10 bytes, but if we get a request for less (say 1 byte) we don't want to use up a full 10 bytes to satisfy it. Instead, we *split* one of the free regions into 1-byte and 9-byte sections.

Conversely, if the 10-byte chunk is allocated, we want to make sure we get

|| 30 free ||

instead of

|| 10 free || 10 free || 10 free ||

To do this, allocators *coalesce* contiguous free memory into one larger chunk.

Tracking The Size Of Allocated Regions

`free` doesn't take a `size`, so the allocator needs to be able to tell this itself. To do so, it stores a header with each block. Thus, a request to allocate `N` bytes results in the allocator searching for `N + sizeof(header)` bytes.

Embedding a Free List

A straightforward implementation of a free list is a linked list, but normally we build these on the heap by calling `malloc`, which is impossible within the allocation library itself. The free list is instead maintained by embedding data in the free blocks: the size of the block and a pointer to the next free block. Thus, just by keeping a head pointer, we can traverse forwards like an ordinary linked list. It's also easy to tell when we need to coalesce in this situation, just by looking at the distance away of the next free block and the size of the newly freed block.

Growing the heap

When we don't have enough memory to satisfy a request the simplest thing we can do is fail and return `NULL`. Allocators typically request more memory from the OS through calls to `sbrk`.

Basic Strategies

It's hard to come up with a generally optimal solution, because different strategies perform differently based on different inputs.

Best fit

Exhaustively searches through all blocks to find the smallest one that is still large enough, and return that. This is expensive, since every request involves searching through the full free list.

Worst fit

In an effort to leave big chunks free rather than lots of little ones (which makes it harder to satisfy requests), worst fit finds the largest block, splits it, and returns the allocated memory. This also involves an exhaustive search, and studies have shown that it leads to high fragmentation.

First fit

First fit finds the first block that is large enough to satisfy a request and uses it. One downside is the beginning of the free list can become polluted with lots of small blocks. We also need to arbitrarily specify an ordering, like address based ordering.

Next fit

To try to avoid the pollution issue of first fit, next fit keeps a pointer to where it stops each search, and picks up at that point instead of always starting at the front of the list. Like first fit, it stops at the first block that fits.

Other Approaches

There are lots of more complex policies than the ones mentioned.

Segregated Lists

If we know we'll have to satisfy a bunch of requests for similar-sized memory, we can try to prepare by keeping blocks of that size ready and forwarding other requests on to a general purpose allocator. One implementation is *slab allocation*, which prepares memory for a bunch of frequently used data types at boot time.

Buddy Allocation

In order to make coalescing easier, *buddy allocation* satisfies each request by viewing all blocks as sized as a power of 2 and recursively dividing the space up to the smallest size possible. Thus, every block has a "buddy", and when its freed, we can check if the buddy is free and coalesce, recursively. The scheme can suffer from internal fragmentation though, since not all requests neatly fit in 2^n sized containers.

Other Ideas

Other allocators try to address scaling problems using fancier data structures like balanced binary trees, etc.