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.

Paging

On component of allocation is dealing with variable-sized chunks of memory and trying to avoid the problem of segmentation. Another, called *paging* deals with fixed-size chunks. Memory is divided into fixed-size chunks called *pages*, and memory can be viewed as an array of fixed sized *page frames*. The goal, like before, is minimizing space and time overheads.

The basic advantage is simplicity. Because all pages have the same size, if a process needs 4 pages it can just be given the first 4 from the free list. Also, it can support better abstractions of the address space. To keep track of everything, the OS keeps a per-process *page table*, which stores *address translations* so we can figure out what physical memory the pages map to. Addressing works similarly to before; virtual addresses are split into a *virtual page number* and an offset. The OS checks which physical frame the virtual page is stored at, along with the offset translates the address.

Where are Page Tables Stored?

Suppose we have 4 KB pages. Then, in a 32 bit address space, that'd leave 20 bits for page numbering, or about a million such pages, *per process*. This would be ridiculously larger still in a 64 bit address space. This rules out using hardware like the MMU. Instead, the table is stored in memory.

What's stored in the Page Table?

The basic function of the page table is mapping virtual page numbers to physical addresses. The simplest way to do this is using a *linear page table*, which can be indexed like an array using a page number. In the table, we several store useful bit of information:

- *Valid bit*: is a translation is valid? (is the page being used?)
- *Protection bits*: what are read/write permissions?
- *Present bit*: is the page in memory, or on disk?
- *Dirty bit*: has it been modified since being brought into memory?
- *Reference bit*: has the page been accessed? Useful for telling what's popular to be kept in memory.

Paging: Also too slow

Consider the process of carrying out `mov 21 %eax` (21 is an address). Using on bits in the address, we first have to translate this into a physical address, involving one lookup. Then, based on the values of bits in the page table, we either perform another lookup to fulfill the request, or return some sort of exception. This takes two lookups, which can be more than twice as expensive as just one.

Paging: Faster Translations (TLBs)

Paging has benefits but requires storing a large amount of memory information (potentially hundreds of megabytes), and decreases performance since resolving any virtual address involves a extra lookup. To address this problem, we introduce new hardware: the *translation-lookaside buffer (TLB).* It's a part of the MMU, and caches virtual address mappings so popular ones can be quickly resolved.

TLB Basic Algorithm

This is normal caching: hardware checks the TLB for the address translation, and if it's there we have a *TLB hit.* Otherwise, we have a *miss*, load it into the *TLB*, and return the value. Based on *spatial and temporal locality* (re-references nearby in space and time), under normal usage with caching our *TLB* hit rate can be substantially improved.

Handling TLB misses

In theory, TLB misses could be resolved in either hardware or software. In early systems this was done in hardware, but on modern systems hardware simply raises an exception, transferring control to a special trap handler. The main advantage is flexibility: hardware has to do very little and it's easy to swap out different structures and handlers for TLB misses.

TLB Contents

TLBs basically store maps from VPNs to PFNs, along with some additional bits to indicate if mapping are *valid*, *protected*, etc. It's a *fully associative cache*, meaning that any given entry can end up anywhere in the cache. Thus, the full cache is search in parallel in hardware.

Context Switches

One problem is that mappings are only valid in the context of a single process, so if the OS preempts it and starts another, everything in the TLB would be invalid. The simple solution is to just flush the TLB for every context switch, but this can be costly. One solution is hardware support to indicate which process an address mapping belongs to.

Replacement Policy

Adding an entry to the TLB cache might involve removing something that's already there. To do this, we employ a policy like *least-recently used (LRU).*

Paging: Smaller Tables

> potentially confusing – read carefully

The problem with array-based page tables is that they take up a lot of memory: potentially hundreds of megabytes in a 32 bit address space and much more in a 64 bit one. An apparent solution to this is just allocating larger pages, but this increases space wastage within pages – *internal fragmentation.*

A better solution is a hybrid approach. One page table can be used per code segment (stack, heap, code) rather than one for the entire process, and we can keep a *bounds* register to keep track of the last allocated page (looking beyond this will raise an exception). This dramatically reduces space wastage. Where we previously might have had lots of unused pages between code segments, this is completely eliminated. However, this approach remains susceptible to fragmentation: a sparse heap could result in lots of wasted space within a page, and page table entries being variable sized reintroduces problems previously encountered when dealing with segmentation.

Multi-level Page Tables

The problem we were trying to get rid of by introducing segmentation was having to keep track of of a large number of invalid page table entries explicitly (just to have an "invalid" bit). Another approach, which doesn't use segmentation, is a tree-like structure (*page table directory*) where we keep track of pages of page table entries, and don't store that full page if none of the entries are valid. Thus, for a given entry we can either tell where it is (if it's valid), or know that it's invalid if it's not there. It's kind of like storing a list of pointers with indicators of which are free to dereference, rather than copies in place. The downsides are a performance hit (two references on a TLB miss), plus a overall increase in complexity.

Inverted Page Tables

Rather than maintaining a page table per process, we could keep a page table per physical page, storing an indication of which process is using it, and the corresponding virtual address. Then, figuring out how to resolve a mapping just resolves to searching using some efficient data structure.

Swapping Page Tables to Disk

When memory gets tight, pages can be *swapped* onto disk.

Beyond Physical Memory: Mechanisms

If we don't make the simplifying assumption that the address space is so small that it can all fit into memory, we have to figure out how to support larger address spaces. Typically, this is done by stashing portions of the address space into larger memory like hard drives. The goal is transparently supporting a larger address space than physical memory. This is especially important in multiprogramming, since we have to manage the address space of much more than one program. Also, it's a much easier-to-user interface for the programmer.

Swap Space

Space allocated on the hard disk for storing pages is called *swap space.* When a request for memory is made, first the TLB (cache) is checked. If it isn't there, then physical memory is checked. If it still isn't there (it could still be in swap space), a *page fault* is raised. This invokes a *page fault handler*, which will usually schedule retrieval from swap space (with the process blocking till the request resolves). If there's no room, the OS will *page out* one or more pages based on some *page replacement policy.* In practice, many operating systems do something just a bit more complicated: they try to keep the amount of free space in memory between some *high and low watermarks*. From a high level, this is the flow of resolving a TLB miss:

1) If the page is *valid* and *present*, we grab the frame number from the page table an retry the request. 2) Otherwise, we generate a page fault. This could load the page. 3) The page could also not be valid, which raises another exception.

And all this happens transparently to the process.

Beyond Physical Memory: Policies

Having a well-designed policy for managing how pages are swapped in and out of memory is crucial because getting it wrong can result in an orders-of-magnitude performance hit. This is a basic *caching* problem: we want to minimize the *average memory access time (AMAT)*. Even small changes in miss rate can have dramatic effects on *AMAT*, since the time cost of accessing disk is orders of magnitude more than accessing memory.

Optimal Policy: Furthest-Out

The optimal policy, which is useful to keep in mind as a point of comparison, would be to evict the thing that's needed furthest in the future.

A Simple Policy: FIFO

FIFO is simple to implement, has no ability to gauge the importance of each page in the cache. When an eviction is necessary, it just kicks out the oldest entry. Performance is much worse than optimal.

Another Simple Policy: Random

Another simple policy is randomly picking what to evict. Performance obviously varies.

Using History: LRU

Based on the *principle of locality*, pages could be evicted based on how recently they were accessed. General reasoning about locality motivates policies like *least recently used (LRU)* and *least frequently used (LFU)*.

The cost of these policies is that they can be expensive to implement: even with hardware support for time-stamping memory accesses, the OS would still have to scan through an array of timestamps. We could easily eat up performance gains performing this sort of maintenance. A solution is approximation: hardware sets a special *use bit* on each page every time its accessed, and the OS periodically clears them. Then, when an eviction is necessary, the first item without its use bit set is evicted. We can also keep a dirty bit, indicating if a page must be written through to be evicted (which is a more expensive than evicting a non-dirty page for free).

Other Policies

Other approaches include pre-fetching pages that have a high change of being required, and clustering writes in batches rather than writing through one at a time.

Thrashing

One final problem is *thrashing* which happens when there just isn't enough memory for the running processes, leading to continues expensive evictions. Some early systems practiced *admission control*, temporarily running a subset of processes. Linux runs an *out-of-memory killer* which randomly selects a process to be terminated.

Complete Virtual Memory Subsystems

> basic end-to-end principles through examination of one early system and Linux

VAX/VMS Virtual Memory

VMS is a good case to look at because it was designed to run on devices of varying power, and also had to deal with certain cases of suboptimal hardware support. VMS systems used the previously mentioned hybrid of segmentation and paging, with a 32-bit address space and 512 byte pages. To avoid excessive memory wastage, the user address space was segmented in two, with a base and bounds register for each. Further, user page tables are stored in kernel virtual memory, making address translation more complicated but unifying page swapping. Also, kernel memory was mapped into the user address spaces, but protected using a bit.

Eviction was not based on recency of reference (no reference bit was kept), but instead aimed to use memory more fairly. This was done using segmented FIFO replacement, where each process could keep a certain number of pages in memory till it had to start evicting them. However, instead of directly writing through evictions, VMS systems kept a global *second chance* list. Page faults would try to read from this before going to disk, and clean pages could be claimed from here. It's basically a extra buffer, without segmentation. Writes could also easily be batched together from the global dirty list.

Two other optimizations from VMS are *demand zeroing* which lazily zeros pages (necessary for security reasons), and *copy-on-write*, which delays copies until a write is performed, keeping a reference to the copied page at first that is read-only and only trapping when a write is attempted.

Linux Virtual Memory

The Linux virtual memory system also separates kernel and process address spaces, with kernel memory being the same for all processes. There are two types of kernel addressed: *logial addresses* which directly map to physical memory (and thus provide significant guarantees), and *virtual addresses* which provides weaker guarantees but are easier to allocate. Most kernel data structures reside in logical memory.

Page Table Structure

The page table is a multi-level structure, with one page table per process. Since the 64 bit address space is much larger than practically necessary, the top 16 bits in virtual addresses go unused, with other used to index into levels of the table. There's also support for *huge pages* which allows for a reduced TLB miss rate, at the cost of potentially more internal fragmentation. Various subsystem data, like file data are also kept in page cache to speed up overall efficiency. The replacement policy is a modified version of LRU: two queues are kept, based on different activity levels, and evictions are done from the inactive queue using approximation methods. There are also various defenses against attacks like buffer overflows.