Making Paging Practical

Andrew Quinn

Learning Objectives:

1. How can we handle large page tables efficiently?

Announcements:

1. Have you started working on Project 2?

We saw from last class how Paging essentially solves external fragmentation because it (1) enforces fixed and equal sized pages and frames, thereby ensuring that *any* page can go to any frame, and (2) allows contiguous virtual memory pages to be placed non-contiguously in physical memory. However, we ended by establishing a tradeoff surrounding page sizes. Namely, if we use large pages, then applications will create a lot of internal fragementation. But, if we small pages, then the operating system will need to manage large page tables.

This lecture will cover a few issues surrounding these relatively large page tables: (1) How should we lay them out? (2) How can we make them faster and (3) Where should we put them? Then, we'll talk a bit about shared frames.

What page table data structure should we use?

Our linear page table representation is too large. Imagine a system with page table entries that are 4 bytes, a 4GB address space, and a page size of 4KiB. Each page table in the system will be 4MiB. Given that the system has a page table *for every* process, and that there are likely hundreds of processes. This is untenable.

Page tables are large hash maps indexed by page number. The page tables for most processes are sparse, since very few programs will use a large portion of a 4GB (or 200+ TiB) virtual address space. Can we take advantage of this sparcity to save room in our page table?

Before we talk about the actual solution, why not use a classic solution for storing hash tables? Namely: a self balancing binary search tree. Why not use one?

There's one additional note about the nature of the sparcity of virtual addresses spaces: applications tend to clump memory together in contiguous chunks. For example, stack pages will be in a contiguous region within the address space, as will heap pages. This suggests that a virtual address space will be made up of alternating contiguous valid pages followed by alternating contiguous invalid pages.

This idea informs an idea called a multi-level page table, based upon the idea of adding a layer of indirection. The OS splits the global page table into small regions, each with its own page table, and uses a *page directory* to keep track of the page tables. Translating a page then becomes a more expensive and complex operation. Given a virtual address, the system must first determine the page directory entry, and then determine the physical frame by indexing into the resulting page table. Finally, it must add in the offset of the virtual address within that page table entry.

What is the tradeoff here?

HOW CAN THE ARCHITECTURE EFFICIENTLY USE PAGE TABLES?

Translating a virtual address is expensive. What is the standard solution to solving a performance problem in computer systems? Caching. The key feature that this caching exploits is that applications exhibit very good temporal locality in the pages that they use.

So, architectures created the translation lookaside buffer, a cache of translations from virtual pages to their cooresponding physical frames. A TLB is quite simple: on translation, check if the virtual page is in the TLB before performing the translation. There's two main questions:

1. Who handles TLB misses? Most architectures will walk the page table based upon a *page table base register*, set by the operating system. Some architecutres (e.g., MIPS) instead trigger a special type of fault and have the operating system fill the TLB. This second design provides freedom for operating systems to implement more interesting caching and data structure schemes for the TLB and page table. But, it tends to be slow.
2. How does the TLB interact with context switches? The problem is that the same virtual page is probably to different frames by different processes. Some architectures requiring "flushing" the TLB on every conetxt switch to handle this. Flushing is expensive, but easy. The other alternative is to introduce an *address space ID* into the TLB so that it can include mapings for multiple processes [1].

WHERE SHOULD WE STORE PAGE TABLES?

Where should we actually store page tables? They have to be accessible by the hardware when doing a translation, which seems to imply that page tables ought to be located in physical memory (i.e., directly stored in some physical frame).

However, even with the space savings of multi-level page tables, they still take up a lot of space. One potential place for further space reduction is that virtual address spaces are not only sparce in terms of the number of valid virtual pages, they are also sparse in terms of the virtual pages that they *actually* use within a window of time. That is: they exhibit temporal locality.

So, can an OS recursively apply paying to page tables? Yep, most systems do. They give the operating system kernel virtual memory and store page tables there. Its worth noting that you cannot store the kernel's page table in kernel virtual memory; "the [translation] buck has to stop somewhere".

SHARED FRAMES

A frame can be transparently shared across different applications, provided that the frame have the *exact* same contents between the two. It is pretty computationally expensive to

[1] But, you should be weary of shared caches!

determine when two frames have the same contents, so it is not common for operating systems to try to share frames in general.

However, there is one Unix system call that is guaranteed to produce many shared frames: fork. We'll go through an example in class. The issue is that some of the process' pages are writable, and we want a process's writes to only apply to themselves. The solution to this issue is what is called *copy-on-write*.