CMPSCI 377 Operating Systems **Fall 2009**

Lecture 15

Lecturer: Mark Corner *Scribes: Bruno Silva,Jim Partan*

15.1 Page Tables

At the end of the last lecture, we introduced *page tables*, which are lookup tables mapping a process' virtual pages to physical pages in RAM. How would one implement these page tables?

15.1.1 Single-Level Page Tables

The most straightforward approach would simply have a single linear array of page-table entries (PTEs). Each PTE contains information about the page, such as its physical page number ("frame" number) as well as status bits, such as whether or not the page is valid, and other bits to be discussed later.

If we have a 32-bit architecture with 4k pages, then we have 2^{20} pages, as discussed in the last lecture. If each PTE is 4 bytes, then each page table requires 4 Mbytes of memory. And remember that each process needs its own page table, and there may be on the order of 100 processes running on a typical personal computer. This would require on the order of 400 Mbytes of RAM just to hold the page tables on a typical desktop!

Furthermore, many programs have a very sparse virtual address space. The vast majority of their PTEs would simply be marked invalid.

Clearly, we need a better solution than single-level page tables.

15.1.2 Multi-Level Page Tables

Multi-level page tables are tree-like structures to hold page tables. As an example, consider a two-level page table, again on a 32-bit architecture with $2^{12} = 4$ kbyte pages. Now, we can divide the virtual address into three parts: say 10 bits for the level-0 index, 10 bits for the level-1 index, and again 12 bits for the offset within a page.

The entries of the level-0 page table are pointers to a level-1 page table, and the entries of the level-1 page table are PTEs as described above in the single-level page table section. Note that on a 32-bit architecture, pointers are 4 bytes (32 bits), and PTEs are typically 4 bytes.

So, if we have one valid page in our process, now our two-level page table only consumes

$(2^{10}$ level-0 entries$) \cdot (2^2$ bytes/entry$) + 1 \cdot (2^{10}$ level-1 entries$) \cdot (2^2$ bytes/entry$) = 2 \cdot 2^{12}$ bytes $= 8$ kbytes.

For processes with sparse virtual memory maps, this is clearly a huge savings, made possible by the additional layer of indirection.

Note that for a process which uses its full memory map, that this two-level page table would use slightly more memory than the single-level page table (4k+4M versus 4M). The worst-case memory usage, in terms of efficiency, is when all 2^{10} level-1 page tables are required, but each one only has a single valid entry.

In practice, most page tables are 3-level or 4-level tables. The size of the indices for the different levels are optimized empirically by the hardware designers, then these sizes are permanently set in hardware for a given architecture.

15.1.3 Page-Table Lookups

How exactly is a page table used to look up an address?

The CPU has a page table base register (PTBR) which points to the base (entry 0) of the level-0 page table. Each process has its own page table, and so in a context switch, the PTBR is updated along with the other context registers. The PTBR contains a physical address, not a virtual address.

When the MMU receives a virtual address which it needs to translate to a physical address, it uses the PTBR to go to the the level-0 page table. Then it uses the level-0 index from the most-significant bits (MSBs) of the virtual address to find the appropriate table entry, which contains a pointer to the base address of the appropriate level-1 page table. Then, from that base address, it uses the level-1 index to find the appropriate entry. In a 2-level page table, the level-1 entry is a PTE, and points to the physical page itself. In a 3-level (or higher) page table, there would be more steps: there are N memory accesses for an N-level page table.

This sounds pretty slow: N page table lookups for *every* memory access. But is it necessarily slow? A special cache called a TLB[1] caches the PTEs from recent lookups, and so if a page's PTE is in the TLB cache, this improves a multi-level page table access time down to the access time for a single-level page table.

When a scheduler switches processes, it invalidates all the TLB entries. The new process then starts with a "cold cache" for its TLB, and takes a while for the TLB to "warm up". The scheduler therefore should not switch too frequently between processes, since a "warm" TLB is critical to making memory accesses fast. This is one reason that *threads* are so useful: switching threads within a process does not require the TLB to be invalidated; switching to a new thread within the same process lets it start up with a "warm" TLB cache right away.

So what are the drawbacks of TLBs? The main drawback is that they need to be extremely fast, fully associative caches. Therefore TLBs are very expensive in terms of power consumption, and have an impact on chip real estate, and increasing chip real estate drives up price dramatically. The TLB can account a significant fraction of the total power consumed by a microprocessor, on the order of 10% or more. TLBs are therefore kept relatively small, and typical sizes are between 8 and 2048 entries.

An additional point is that for TLBs to work well, memory accesses have to show *locality*, i.e. accesses aren't made randomly all over the address map, but rather tend to be close to other addresses which were recently accessed.

15.1.4 Introduction to Paging

We will briefly introduce *paging* to finish off this lecture. When a process is loaded, not all of the pages are immediately loaded, since it's possible that they will not all be needed, and memory is a scarce resource. The process' virtual memory address space is broken up into pages, and pages which are valid addresses are either loaded or not loaded. When the MMU encounters a virtual address which is valid but not loaded (or "resident") in memory, then the MMU issues a *pagefault* to the operating system. The OS will then load the page from disk into RAM, and then let the MMU continue with its address translation. Since access times for RAM are measured in nanoseconds, and access times for disks are measured in milliseconds, excessive pagefaults will clearly hurt performance a lot.

[1] TLB stands for "translation lookaside buffer", which is not a very enlightening name for this subsystem.