

Page Table Notes

Rijurekha Sen

April 2026

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?

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, or "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 4KB pages, then we have 2^{20} pages. If each PTE is 4 bytes, then each page table requires 4MB of memory. 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 400MB of RAM just to hold the page tables.

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.

1.2 Multi-Level Page Tables

Multi-level page tables are tree-like structures used to hold page tables. As an example, consider a two-level page table on a 32-bit architecture with $2^{12} = 4\text{KB}$ pages. We can divide the virtual address into three parts: 10 bits for the level-0 index, 10 bits for the level-1 index, and 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. On a 32-bit architecture, pointers are 4 bytes, and PTEs are typically 4 bytes.

If we have one valid page in our process, the two-level page table consumes only

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

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

For a process that uses its full memory map, the two-level page table would use slightly more memory than the single-level page table: 4KB + 4MB versus 4MB. The worst-case memory usage, in terms of efficiency, occurs when all 2^{10} level-1 page tables are required, but each one has only a single valid entry.

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

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, or entry 0, of the level-0 page table. Each process has its own page table, so during 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 that it needs to translate to a physical address, it uses the PTBR to access the level-0 page table. It then 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. 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 slow: N page-table lookups for every memory access. But is it necessarily slow? A special cache called a TLB¹ caches the PTEs from recent lookups. If a page's PTE is in the TLB, this improves 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 TLB entries. The new process then starts with a “cold cache” for its TLB, and it takes time for the TLB to “warm up.” Therefore, the scheduler 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 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 with a warm TLB cache right away.

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

For TLBs to work well, memory accesses must show locality. That is, accesses should not be made randomly all over the address map, but should instead tend to be close to other addresses that were recently accessed.

¹TLB stands for “translation lookaside buffer.”

1.4 Introduction to Paging

We briefly introduce paging to finish this lecture. When a process is loaded, not all of its pages are immediately loaded, since it is possible that they will not all be needed, and memory is a scarce resource.

The process' virtual memory address space is broken into pages. Pages that are valid addresses are either loaded or not loaded. When the MMU encounters a virtual address that is valid but not loaded, or "resident," in memory, the MMU issues a page fault to the operating system.

The OS then loads the page from disk into RAM and lets the MMU continue with its address translation. Since access times for RAM are measured in nanoseconds, while access times for disks are measured in milliseconds, excessive page faults can hurt performance significantly.