ECE 344: Operating Systems

Lecture 20

Multi-Level Page Tables

1.0.4

Jon Eyolfson

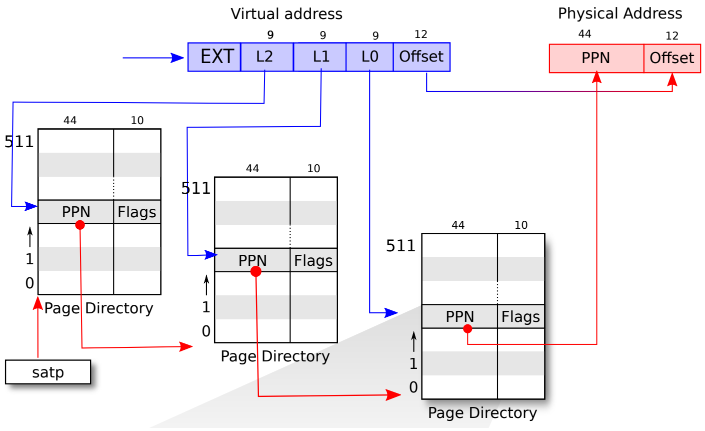
October 27, 2021

Please Provide Your Feedback

Fill out the following form and help improve the course!

`https://forms.gle/SxUoFYWimDwNoNcs9`

Multi-Level Page Tables Save Space for Sparse Allocations



© MIT `https://github.com/mit-pdos/xv6-riscv-book/`

For RISC-V Each Level Occupies One Page

There are 512 (2^9) entries of 8 bytes(2^3) each, which is 4096 bytes

The PTE for L(N) points to the page table for L(N-1)

You follow these page tables until L0 and that contains the PPN

Consider Just One Additional Level

Assume our process uses just one virtual address at `0x3FFFF008`

  or `0b11_1111_1111_1111_1111_0000_0000_1000`

  or `0b111111111_111111111_000000001000`

We'll just consider a 30-bit virtual address with a page size of 4096 bytes

  We would need a 2 MiB page table if we only had one ($2^{18} \times 2^{3}$)

Instead we have a 4 KiB L1 page table ($2^{9} \times 2^{3}$) and a 4 KiB L0 page table

  Total of 8 KiB instead of 2 MiB

Note: worst case if we used all virtual addresses we would consume 2 MiB + 4 KiB

Translating 3FFFF008 with 2 Page Tables

Consider the L1 table with the entry:

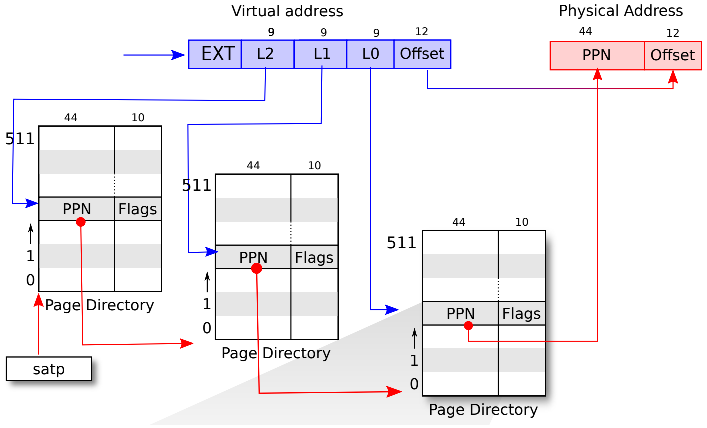
Index	PPN
511	0x8

Consider the L0 table located at 0x8000 with the entry:

Index	PPN
511	0xCAFE

The final translated physical address would be: 0xCAFE008

Processes Use A Register Like satp to Set the Root Page Table



© MIT https://github.com/mit-pdos/xv6-riscv-book/

Page Allocation Uses A Free List

Given physical pages, the operating system maintains a free list (linked list)

The unused pages themselves contain the `next` pointer in the free list

- Physical memory gets initialized at boot

To allocate a page, you remove it from the free list

- To deallocate a page you add it back to the free list

Using the Page Tables for Every Memory Access is Slow

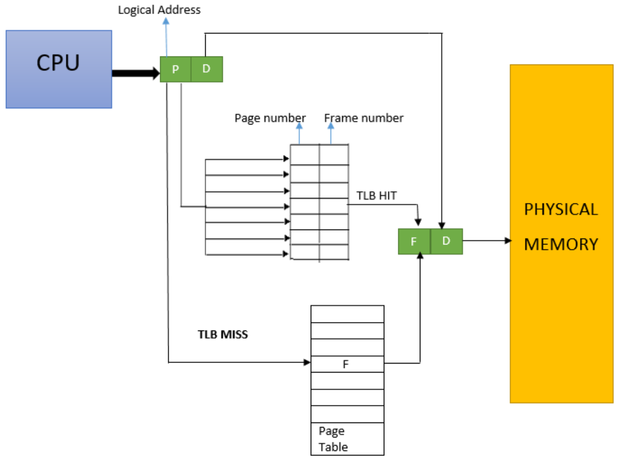
We need to follow pointers across multiple levels of page tables!

We'll likely access the same page multiple times (close to the first access time)

A process may only need a few VPN $\rightarrow$ PPN mappings at a time

Our solution is another computer science classic: caching

A Translation Look-Aside Buffer (TLB) Caches Virtual Addresses



“Working flow of a TLB” by Aravind Krishna is licensed under CC BY-SA 4.0

Effective Access Time (EAT)

Assume a single page table (there's only one additional memory access in the page table)

$$\text{TLB\_Hit\_Time} = \text{TLB\_Search} + \text{Mem}$$
$$\text{TLB\_Miss\_Time} = \text{TLB\_Search} + 2 \times \text{Mem}$$
$$\text{EAT} = \alpha \times \text{TLB\_Hit\_Time} + (1 - \alpha) \times \text{TLB\_Miss\_Time}$$

If $\alpha = 0.8$, $\text{TLB\_Search} = 10$ ns, and memory accesses take 100 ns, calculate EAT

$$\text{EAT} = 0.8 \times 110 \text{ ns} + 0.2 \times 210 \text{ ns}$$
$$\text{EAT} = 130 \text{ ns}$$

Context Switches Require Handling the TLB

You can either flush the cache, or attach a process ID to the TLB

Most implementation just flush the TLB

- RISC-V uses a `sfence.vma` instruction to flush the TLB

On x86 loading the base page table will also flush the TLB

How Many Levels Do I Need?

Assume we have a 32-bit virtual address with a page size of 4096 bytes

and a PTE size of 4 bytes

We want each page table to fit into a single page

Find the number of PTEs we could have in a page (2^{10})

$\log_2(\#\text{PTEs per Page})$ is the number of bits to index a page table

$$\#\text{Levels} = \left\lceil \frac{\text{Virtual Bits} - \text{Offset Bits}}{\text{Index Bits}} \right\rceil$$

How Many Levels Do I Need?

Assume we have a 32-bit virtual address with a page size of 4096 bytes
 and a PTE size of 4 bytes

We want each page table to fit into a single page
 Find the number of PTEs we could have in a page (2^{10})
  $\log_2(\#\text{PTEs per Page})$ is the number of bits to index a page table

$$\#\text{Levels} = \left\lceil \frac{\text{Virtual Bits} - \text{Offset Bits}}{\text{Index Bits}} \right\rceil$$

$$\#\text{Levels} = \left\lceil \frac{32-12}{10} \right\rceil = 2$$

TLB Testing

Check out `lecture-20/test-tlb`

(you may need to `git submodule update --init --recursive`)

`./test-tlb <size> <stride>`

Creates a <size> memory allocation and acccesses it every <stride> bytes

Results from my laptop:

```
> ./test-tlb 4096 4
  1.93ns (~7.5 cycles)
> ./test-tlb 536870912 4096
155.51ns (~606.5 cycles)
> ./test-tlb 16777216 128
 14.78ns (~57.6 cycles)
```

Use `sbrk` for Userspace Allocation

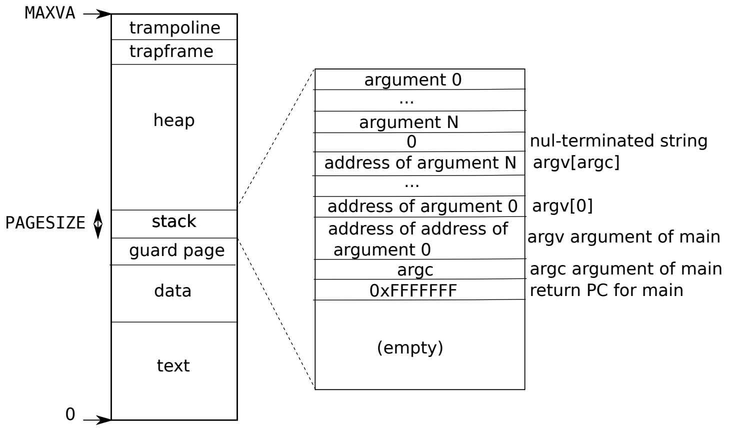
This call grows or shrinks your heap (the stack has a set limit)

For growing, it'll grab pages from the free list to fulfill the request
- The kernel sets PTE_V (valid) and other permissions

In memory allocators this is difficult to use, you'll rarely shrink the heap
- It'll stay claimed by the process, and the kernel cannot free pages

Memory allocators use `mmap` to bring in large blocks of virtual memory

The Kernel Initializes the Processs’ Address Space (and Stack)



The guard page will generate an exception if accessed meaning stack overflow

The Kernel Can Provide Fixed Virtual Addresses

It allows the process to access kernel data without using a system call

For instance `clock_gettime` does not do a system call

- It just reads from a virtual address mapped by the kernel

Page Faults Allow the Operating System to Handle Virtual Memory

- Page faults are a type of exception for virtual memory access
    - Generated if it cannot find a translation, or permission check fails
- This allows the operating system to handle it
    - We could lazily allocate pages, implement copy-on-write, or swap to disk

Page Tables Translate Virtual to Physical Addresses

The MMU is the hardware that uses page tables, which may:

- Be a single large table (wasteful, even for 32-bit machines)
- Be a multi-level to save space for sparse allocations
- Use the kernel allocate pages from a free list
- Use a TLB to speed up memory accesses