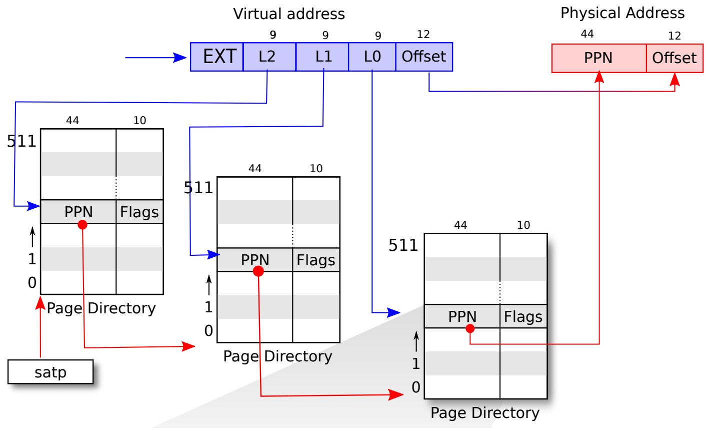
ECE 353: Systems Software
Lecture 15

Page Table Implementation

1.1.0

Jon Eyolfson
February 9, 2023

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



Alignment: Memory Eventually Lines Up with Byte 0

If pages are *4096 byte aligned* in memory is means pages always start when the lower 12 bits are zero, in computing we like alignment

If a page started at address `0x7C00` its last byte would be at address `0x8BFF`

Instead a page would start at `0x7000` and end at `0x7FFF`

Question: Is address `0xEC` 8 byte aligned?

Let’s Simulate an MMU

`14-page-tables` in your `examples` repository

Remember each process would have it’s own unique root page table

How Many Page Tables Do We Need?

Let’s assume our program uses 512 pages

What’s the minimum number of page tables we need?

What’s the maximum number of page tables?

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

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

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

Using the Page Tables for Every Memory Access is Slow

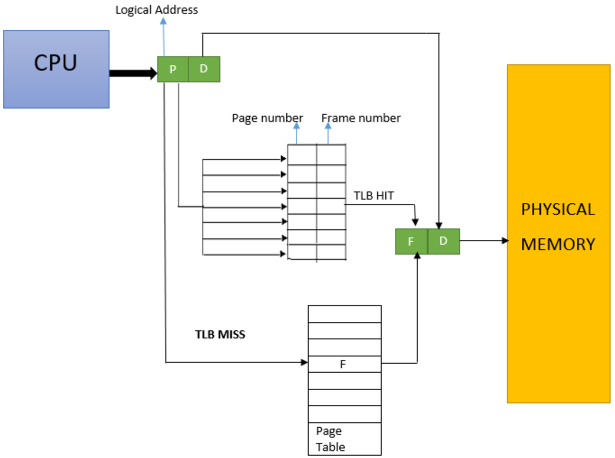
We need to follow pointers across multiple levels of page tables!

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

A process may only need a few VPN $\longrightarrow$ 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

We paused here on Thursday Feb 9

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

TLB Testing

Check out `15-page-table-implementation/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

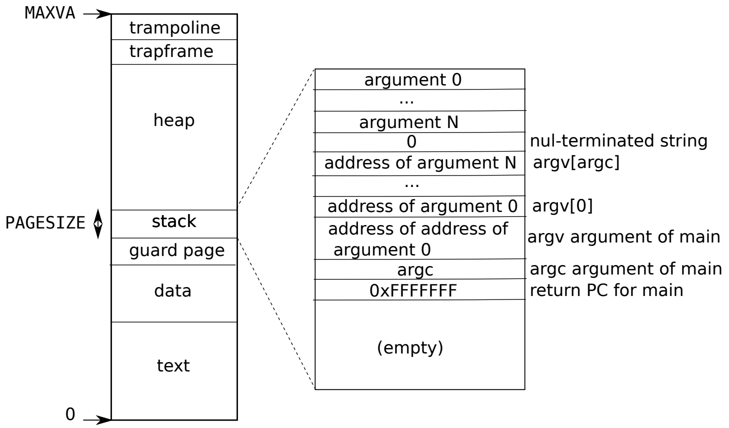
- The kernel sets `PTE_V` (valid) and other permissions

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)
- Use the kernel allocated pages from a free list
- Be a multi-level to save space for sparse allocations
- Use a TLB to speed up memory accesses