CS 111: Operating System Principles

Lecture 10

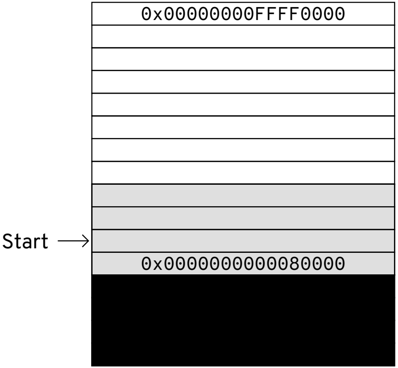
Page Tables

1.0.1

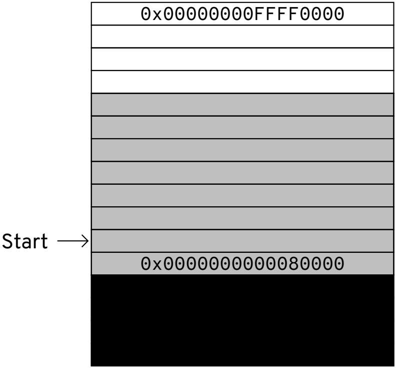
Jon Eyolfson

April 20, 2021

Virtualization Fools Something into Thinking it Has All Resources



“LibreOffice Memory”



“Firefox Memory”

Virtual Memory Checklist

- ☐ Multiple processes must be able to co-exist
- ☐ Processes are not aware they are sharing physical memory
- ☐ Processes cannot access each others data (unless allowed explicitly)
- ☐ Performance close to using physical memory
- ☐ Limit the amount of fragmentation (wasted memory)

Memory Management Unit (MMU)

Maps virtual address to physical address
- Also checks permissions

One technique is to divide memory up into fixed-size pages (typically 4096 bytes)
- A page in virtual memory is called a page
- A page in physical memory is called a frame

Segmentation or Segments are Coarse Grained

Divide the virtual address space into segments for: code, data, stack, and heap

- Note: this looks like an ELF file, large sections of memory with permissions

Each segment is a variable size, and can be dynamically resized

- This is an old legacy technique that's no longer used

Segments can be large and very costly to relocate

- It also leads to fragmentation (gaps of unused memory)

No longer used in modern operating systems

Segmentation Details

Each segment contains a: base, limit, and permissions

You get a physical address by using: `segment selector:offset`

The MMU checks that your offset is within the limit (size)

If it is, it calculates base + offset, and does permission checks

Otherwise, it's a segmentation fault

For example 0x1:0xFF with segment 0x1 base = 0x2000, limit = 0x1FF

Translates to 0x20FF

Note: Linux sets every base to 0, and limit to the maximum amount

You Typically Do Not Use All 64 Virtual Address Bits

CPUs may have different levels of virtual addresses you can use

- Implementation ideas are the same

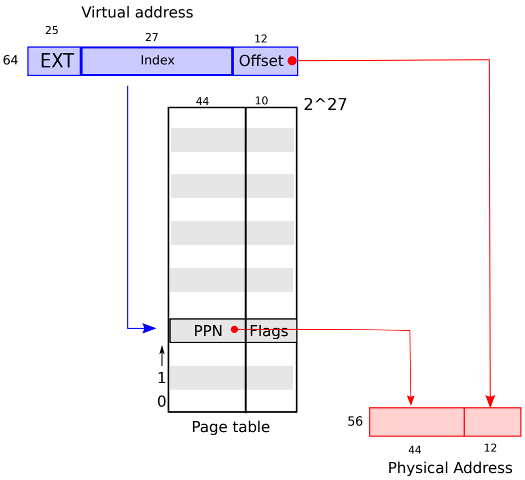
We'll assume a 39 bit virtual address space used by RISC-V and other architectures

- Allows for 512 GiB of addressable memory (called Sv39)

Implemented with a page table indexed by Virtual Page Number (VPN)

- Looks up the Physical Page Number (PPN)

The Page Table Translates Virtual to Physical Addresses



The Kernel Handles Translating Virtual Addresses

Considering the following page table:

VPN	PPN
`0x0`	`0x1`
`0x1`	`0x4`
`0x2`	`0x3`
`0x3`	`0x7`

We would get the following virtual $\rightarrow$ physical address translations:

`0x0AB0` $\rightarrow$ `0x1AB0`

`0x1FA0` $\rightarrow$ `0x4FA0`

`0x2884` $\rightarrow$ `0x3884`

`0x32D0` $\rightarrow$ `0x72D0`

Page Translation Example Problem

Assume you have a 8-bit virtual address, 10-bit physical address and each page is 64 bytes

- How many virtual pages are there?
- How many physical pages are there?
- How many entries are in the page table?
- Given the page table is `[0x2, 0x5, 0x1, 0x8]` what's the physical address of `0xF1`?

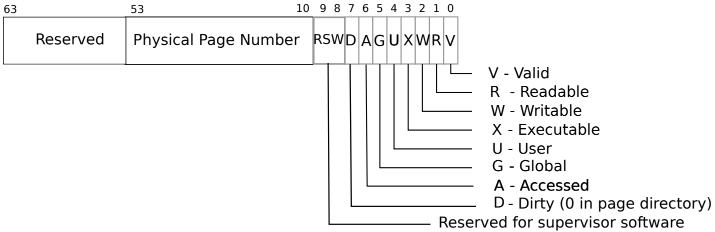
Page Translation Example Problem

Assume you have a 8-bit virtual address, 10-bit physical address and each page is 64 bytes

- How many virtual pages are there? $\frac{2^8}{2^6} = 4$
- How many physical pages are there? $\frac{2^{10}}{2^6} = 16$
- How many entries are in the page table? 4
- Given the page table is `[0x2, 0x5, 0x1, 0x8]` what's the physical address of `0xF1`?

  `0x231`

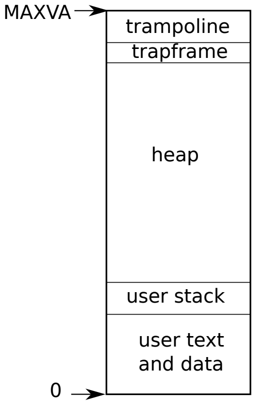
The Page Table Entry (PTE) Also Stores Flags in the Lower Bits



The MMU which uses the page table checks these flags

- We’ll focus on the first 5 flags

Each Process Gets Its Own Virtual Address Space



Each Process Gets Its Own Page Table

When you `fork` a process, it will copy the page table from the parent

- Turn off the write permission so the kernel can implement copy-on-write

The problem is there are 2^{27} entries in the page table, each one is 8 bytes

- This means the page table would be 1 GiB

Note that RISC-V translates a 39-bit virtual to a 56-bit physical address

- It has 10 bits to spare in the PTE and could expand
- Page size is 4096 bytes (size of offset field)

You May Be Thinking That Seems Like A Lot of Work

In Lab 1, we're doing a `fork` followed by `exec` why do we need to copy the page tables?

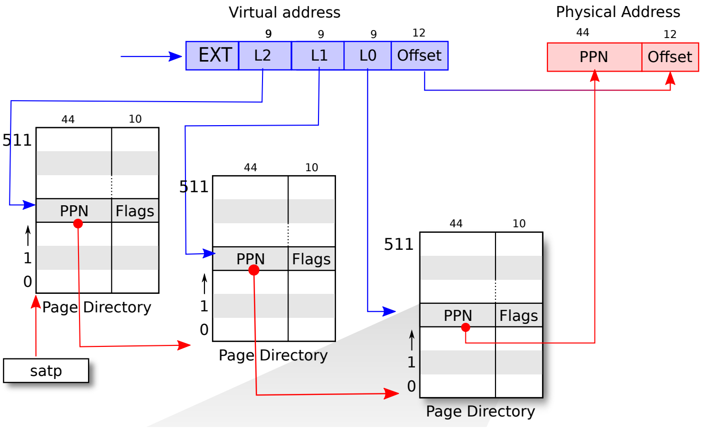
We don't! There's a system call for that – `vfork`

`vfork` shares all memory with the parent

- It's undefined behavior to modify anything

Only used in very performance sensitive programs

Multi-Level Page Tables Save Space for Sparse Allocations



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:

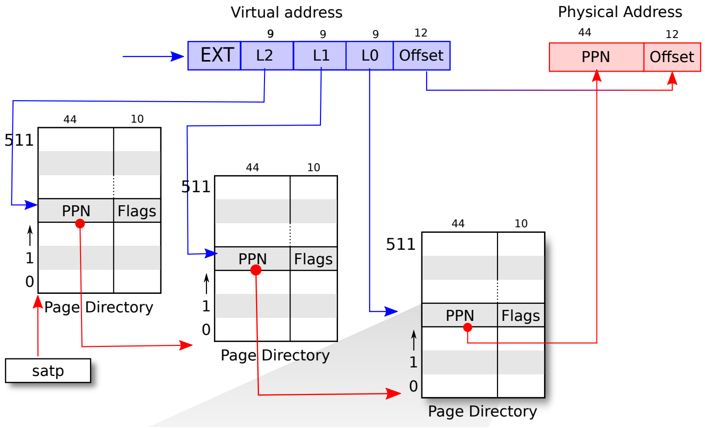
Index	PPN
511	0x8

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



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

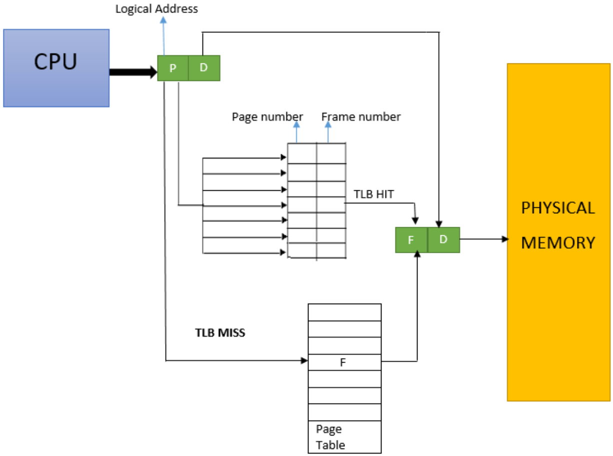
We need to follow pointers across multiple levels of page tables!

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



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-10/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

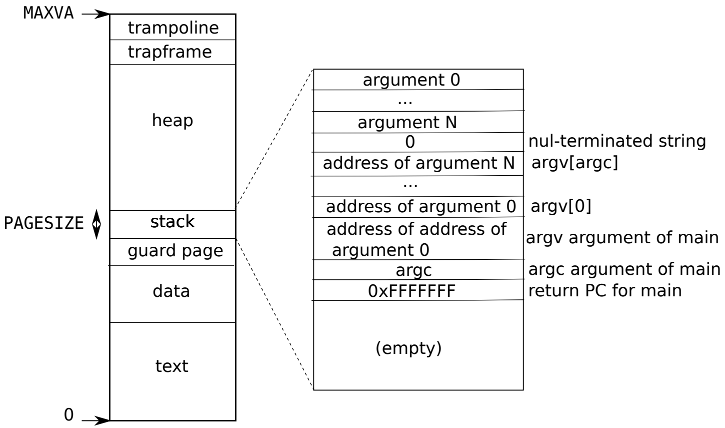
- The kernel sets `PTE_V` (valid) and other permissions

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)



A Trampoline is A Fixed Virtual Address Set by the Kernel

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

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

A trap is anytime special handler code runs:

- System call
- Exception
- Interrupt (e.g timer)

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