ECE 344: Operating Systems

Lecture 19

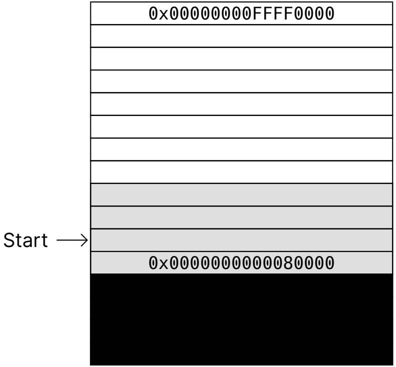
Single-Level Page Tables

1.0.1

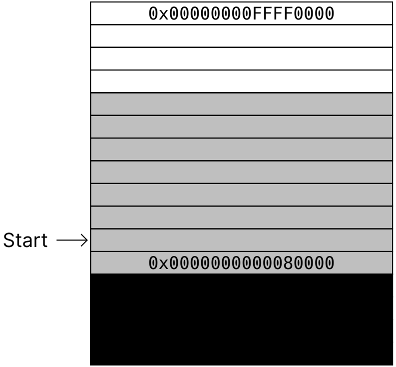
Jon Eyolfson

October 24, 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 0×1:0xFF with segment 0×1 base = 0×2000, limit = 0×1FF

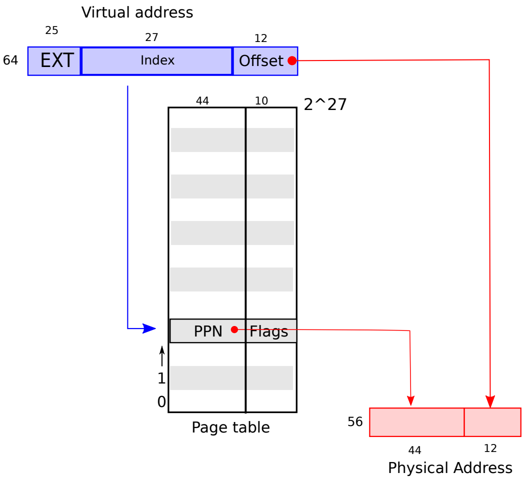
- Translates to 0×20FF

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`?

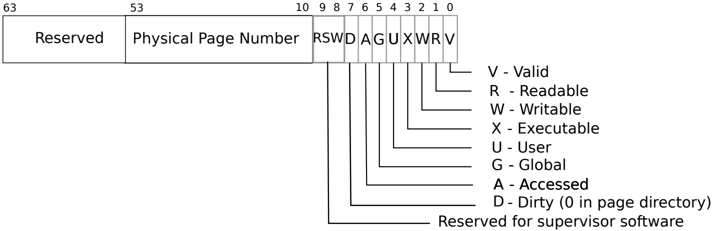
Page Translation Example Problem

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`

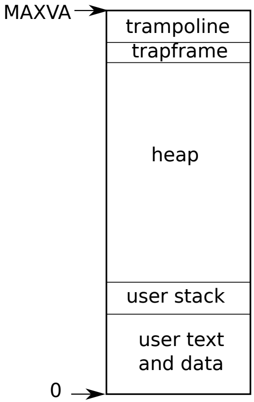
The Page Table Entry (PTE) Also Stores Flags in the Lower Bits



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 the Lab 4 Primer, 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