Recitation 10: Virtual Memory

Griffin Hurt
Undergraduate Teaching Fellow
griffhurt@pitt.edu
https://griffinhurt.com

Spring 2024, Term 2244
Friday 2 PM Recitation
Apr 5th, 2024

Slides adapted from
Shinwoo Kim, Martha Dixon, and Vinicius Petrucci

Department of Computer Science
School of Computing & Information
University of Pittsburgh

- Course News
- Paging/Page Tables
- Quiz
- Work Time

Course News

Shell Project due Tuesday April 9th at 5:59PM

Paging and Page Tables

The Illusion

- Have you ever noticed something about the addresses of your pointers?
  - How heap pointers have their own similar prefixes to their address?
  - And how stack pointers do too?
  - Even if multiple processes are open at once?
- Well, if they were able to all access physical memory, this would lead to *many* collisions, each process overwriting each other’s memory
  - So, to counteract this, we created *virtual* memory
  - An illusion of continuous memory

The Reality

- In reality, physical memory is very similar to a heap, and **malloc**
  - Only, all block sizes are exactly the same - 1 page
- Each process can malloc more pages, and with some magic, the operating system will convert that page’s address in *physical* memory to a *virtual* address
  - This allows each process to believe that they own all of RAM
  - Which they very much *do not*

The Magic

- How does this magical process happen? How are these virtual addresses converted to real addresses?
- Well, we encode information into our virtual address, leaving a breadcrumb trail as to where the real page lies in memory
- This breadcrumb trail is called a page table

The ~~Magic~~ Page Table

- The first thing we need to do is figure out: What is a page?
  - A page is basically an array of bytes
  - The arrays are all of a fixed size, which is just the page size
- So, part of the address we're trying to access should be an index into this large byte array
  - Or the page **offset**
- How do we determine how large of an offset we can have?
  - Well, we need to be able to index our byte array from 0 to page_size - 1
  - If we consider **unsigned numbers**, this looks a lot like the max value for an n bit signed number… where 2^n = page_size
  - See slide 25 in 2's complement for where this came from

The ~~Magic~~ Page Table

- If we say that 2^n = page_size, how do we determine what the page size actually is?
  - Well, we can use logarithms
  - $2^n = page_size \rightarrow n = \log_2(page_size)$
  - This tells us how many **bits** we need to represent the range of indices in the page, namely 0 to page_size - 1

The ~~Magic~~ Page Table

- So, the number of bits that we need for the offset is $\log_2(\text{page_size})$
  - From there, we know where we want to go in the page
  - But… we still don’t know which page we’re trying to access in RAM (physical memory)
- How do we find out which page in RAM we’re trying to access?
- We index our page table!

The ~~Magic~~ Page Table

- The page table is an array of pages
    - More accurately, an array of page *addresses*
        - Which point to RAM (the *physical* address)
- Like any other array, we need to index the page table to access the physical address that the page resides at

The ~~Magic~~ Page Table

- We said earlier that we need some number of offset bits in the virtual address
- Those bits aside, the rest can be used to index the page table to find our page
- So, we’ll mask out the rest of the bits not in the offset, and use them to index the page table
- After we get the real page address in RAM by indexing the page table, we attach the offset to that address (since the index of our page that we’re trying to read won’t change)
    - Now, we have our physical address!
- Let’s look at an example

Paging Example

- We need to know some things about architecture
  - Assume 36 KiB page size
  - Assume 1 GiB RAM
  - Assume we’re using 32 bit architecture
    - Meaning our address size is 32 bits

Paging Example

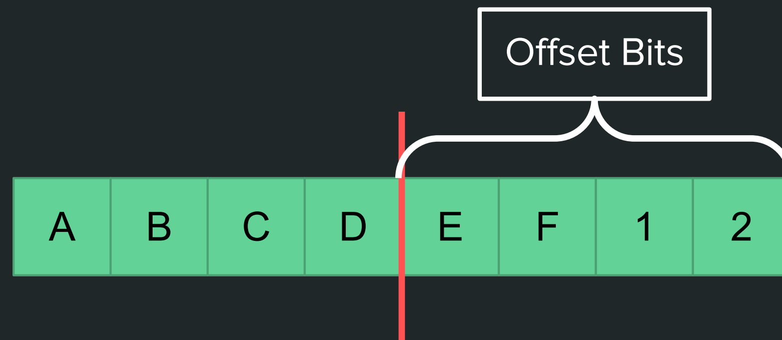
- Number of offset bits:
  - $\log_2(64 \text{ KiB}) = \log_2(2^{16}) = 16$
- Bits used to index the page table:
  - 32 (address size) - 16 (offset bits) = 16
- So, we know that we can have 16 bits to index the page table, or 2^{16} different pages in the page table

Paging Example

A	B	C	D	E	F	1	2

This is our 32 bit address (in hexadecimal)

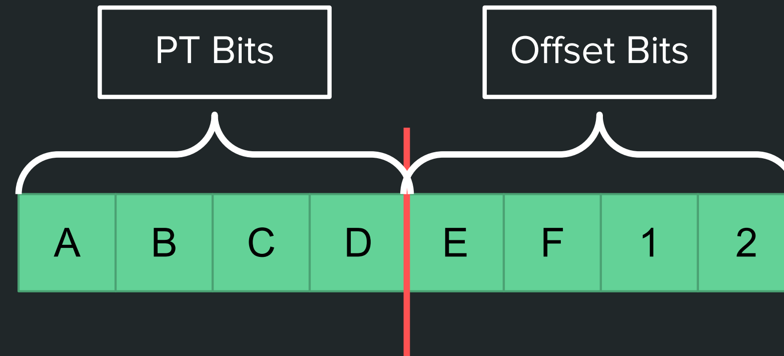
Paging Example



This is our 32 bit address (in hexadecimal)

We know that the lower 16 bits are used for the offset

Paging Example

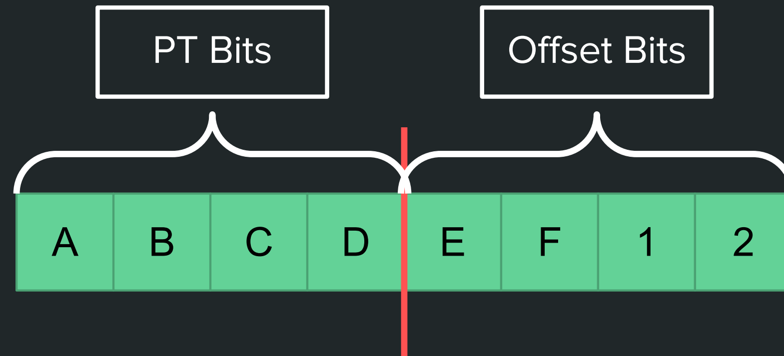


This is our 32 bit address (in hexadecimal)

We know that the lower 16 bits are used for the offset

That means the remaining 16 bits are used for the page table index

Paging Example

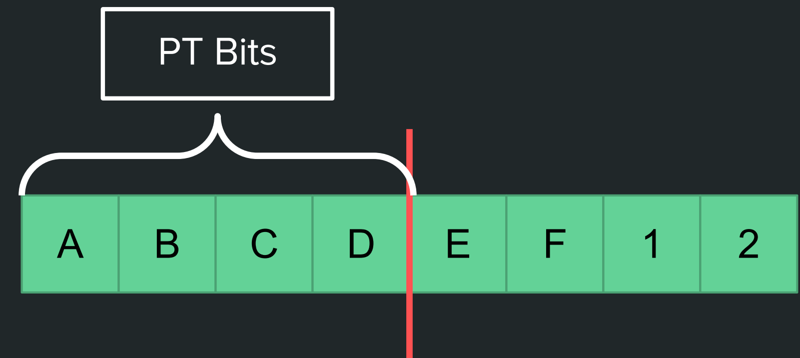


Next, we'll go to the page table, and look at index 0xABCD, since that's the page table index for this virtual address.

Once we index the page table, it will return to us the real, physical address for this page that we're looking for.

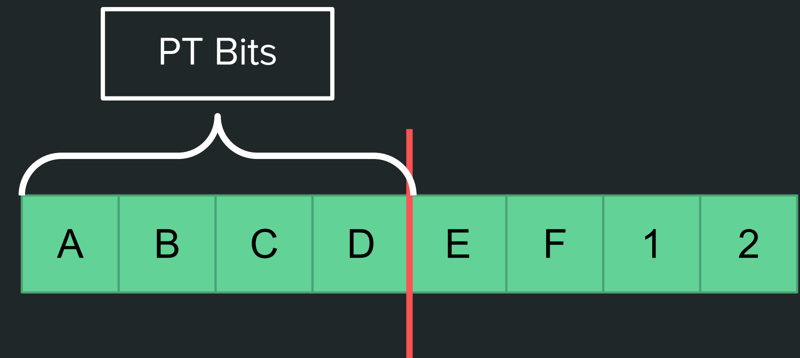
Paging Example

	Valid	Read	Write	Exec	Addr
0x0000	0	0	1	0	0xA
0x0001	0	1	0	1	0x1
0x0002	0	0	1	1	0xF
…	…	…	…	…	…
0xABCC	1	1	0	0	0x7
0xABCD	1	1	1	0	0x3
0xABCE	1	1	1	0	0xC
…	…	…	…	…	…



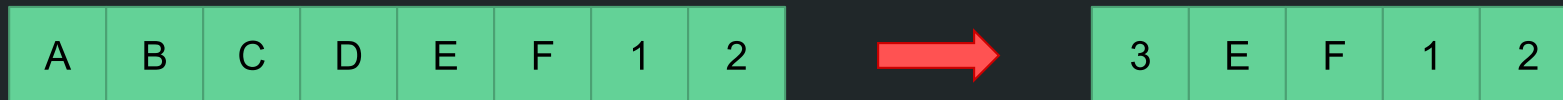
Paging Example

	Valid	Read	Write	Exec	Addr
0x0000	0	0	1	0	0xA
0x0001	0	1	0	1	0x1
0x0002	0	0	1	1	0xF
...	...	...	...	...	...
0xABCC	1	1	0	0	0x7
0xABCD	1	1	1	0	0x3
0xABCE	1	1	1	0	0xC
...	...	...	...	...	...



Page is valid, so we can look for the corresponding physical address, which is 0x3

Paging Example



After going to the page table, we found that the index in physical memory for this page was 0x3, so the real, physical address is now shown on the right.

Quiz time!

Password: ______