Today:

- VM for performance

What was the point of the midterm?

- Hypothesis
    - Dictionary definition
    - Synonym: a guess
- There are multiple ways a talented programmer can understand code
    - One can try and understand everything in the code, but this is very slow
    - Faster way to figure out what's going on with code: make a hypothesis on the code, and then test the guess
    - Testing your ability to guess what the right answer is and then test it
    - Approaching debugging from a scientific method
- Example: Assembly problem on the midterm
    - Which function uses which data structure?
    - Straightforward, bottom-up, step-by-step method: Look at every function and figure out what they do, and work backwards to C code, and then figure out the data structure
    - Quick and dirty way to develop a hypothesis: The four data structures are of varying complexity. Hypothesis: the length.complexity of the assembly code is related to the data structure. Ordering the functions by length (i.e. complexity) and matching to the structures should give the right answers.
    - Shortest code: f4, then f2, then f1 (linked lists), then f3 (binary tree) as the longest
    - How do we confirm this hypothesis? Ask a question that we can then answer.
        - Some of these structures need loops
            - Linked lists, binary tree
        - Loops require branches
        - So look for branches. If the assembly code with branches matches up with what we link are the linked list and binary tree, then it is a confirmation of our hypothesis
            - It confirms
        - Would also confirm hypothesis if array code contains one indirect memory reference (f4). But it seems like the code has three indirect references. But, the first two statements are just loading arguments! So there's only one after that's loading the array value, so we're fine.
- Speed and familiarity with making these kinds of hypotheses should be encouraged, and that's part of what was being tested on the midterm.

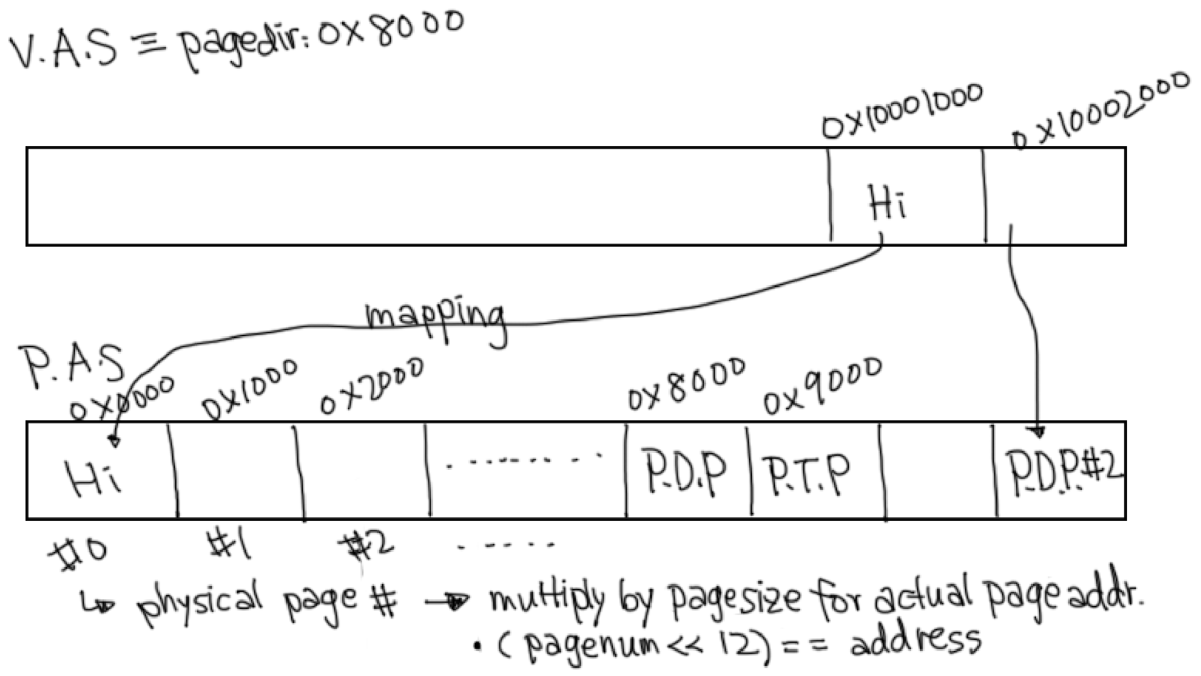
VM worksheet, hex, etc.

- Who has done the VM worksheet?
    - One person...
- YouTube video: Cisco CCNA - Hexadecimal Made Easy - Part 1
    - You can learn about hexadecimal to binary conversion online

    - Master binary math!
- Everyone wants to be a Cisco certified engineer.
    - CCNA exam

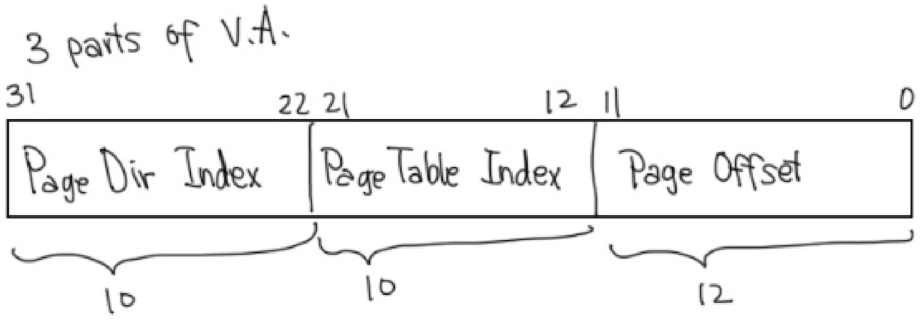
Finishing up VM

- Last time: purpose of some of the two-level page tables
- Now: going through more examples of two-level page tables
- Purpose of VM and two-level page tables: to map virtual address space to physical address space

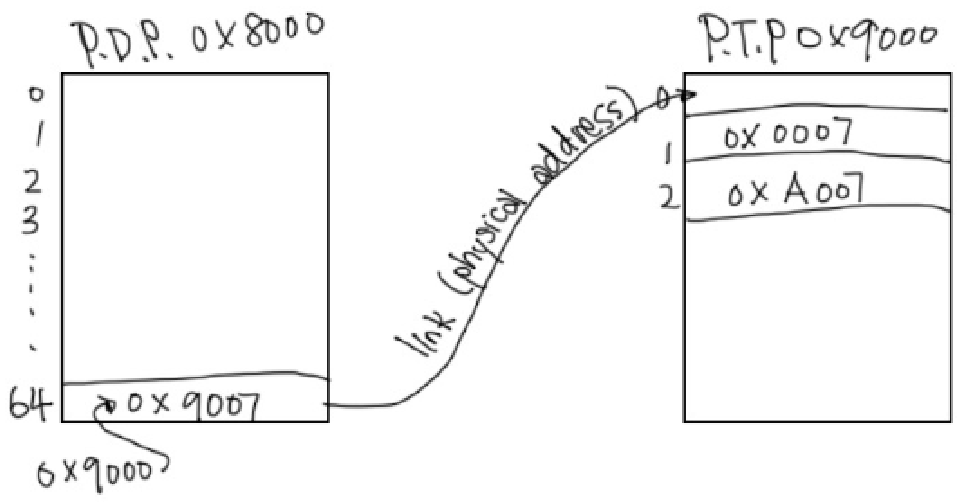


- How big is a page?
    - PAGESIZE = 4096 = 2^{12} = 0x1000
- In x86, all the physical and virtual memory is divided up into units of aligned measurements (pages0
- How do I go from a physical page number to a physical address?
    - Multiply by the page size
    - Or equivalently, since this is a power of 2, we can do a bitshift
    - (pagenum << 12) == address
- Let's put page directory page (PDP) at address 0x8000 and page table page (PTP) at address 0x9000
- In Weensy OS, there's always at least one page directory (the kernel), and as you move through, you make more
- Let's say the physical memory has address 0x8000
- GOAL #1: To map virtual address (VA) 0x10001456 to physical address (PA) 0x00000456
- In order to do this, we need to map out the contents of both the PDP and the PTP

- Each PTP and PDP are arrays of entries; each entry is 4B long
- So there are 1024 (=2^{12}/4) entries in a PTP or PDP
- Last piece of information:



- Three parts of a VA. Bottom 12 bits are the page offset. Top 10 bits are PDI (PDP index), middle 10 are the PTI (PTP index)
- What directory in the PDP do we need to change?
- we are trying to get the top 10 bits, so that's a right shift by 22. to model that, we have a rightshift by 20 then by 2. Rightshift by 20 means we drop 5 digits in hex -> 0x100 = 256. Then 256/4 = 64.
- In hex, rightshift by 4 is like dropping one digit, by 8 is like dropping 2, etc.
- So we need to change entry #64
    - to 0x9000

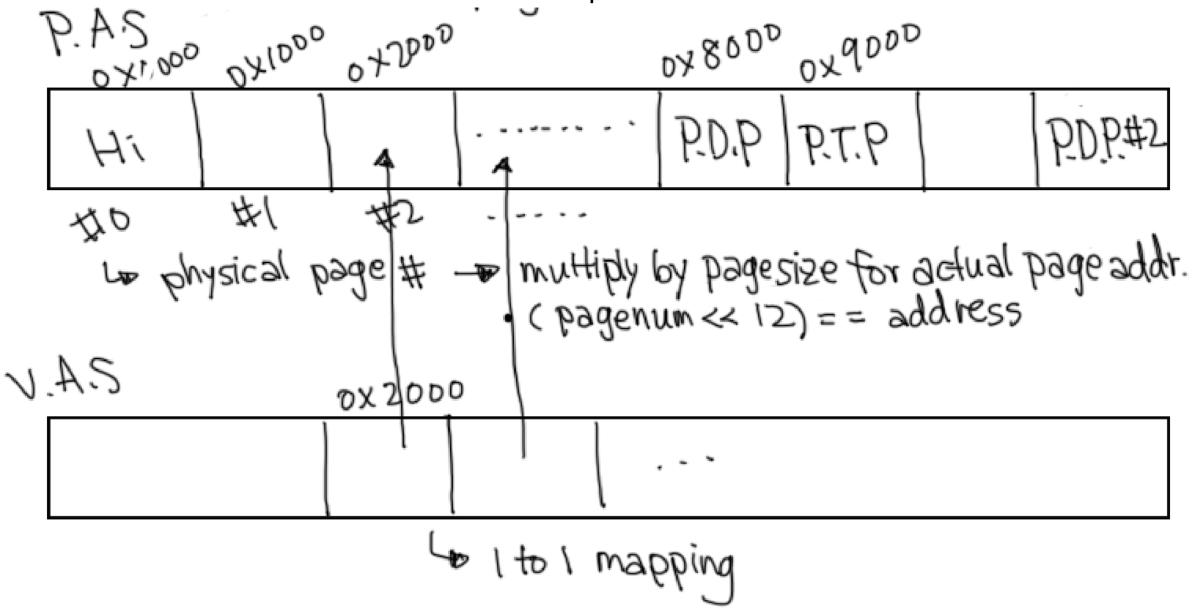


- Flags determine whether a page entry is actually useable (P, W, U)
    - present
    - writable
    - accessible to user processes
- So we now have a link from the PDP to the PTP

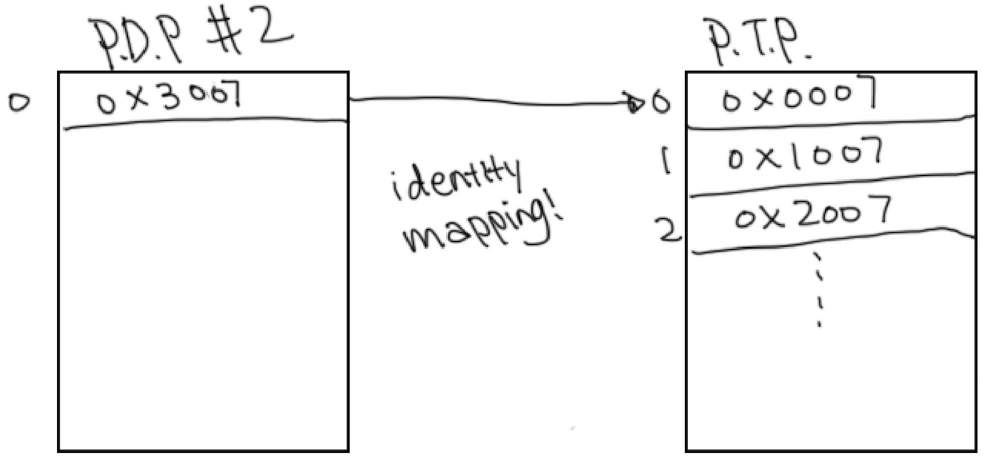
    - But this is a funny link because it's using physical addresses, because it's being used by the memory system in the process of changing physical to virtual addresses
- Which entry in the PTP to we modify to complete this mapping?
    - The index of the PTP -- index 1
- We set it to 0, so with a flag, we make it 7
    - 0x007
- So the virtual address of 0x10001000 corresponds to the physical address 0x0000
- Let's say we wanted to add this mapping:
    - VA: 0x1000149A to PA: 0x0000049A
    - It's the same! Because everything is the same except for the offset [49A]
    - And virtual memory applies the offset after it's gone through the table
- Flags
    - Flags in page entries define <u>permissions</u>.
    - If the PTE_P flag is present, then the page exists.
        - (So if PTE_P is not present, the page doesn't exist)
        - like a null pointer defined by a single bit
        - THe value of the PRE_P flag is 1
    - If the PTE_W flag is present, then the page is writable
        - The value of that flag is 2
        - What does it mean for it to be writable?
            - For process execution, a page may not write to the kernel. So this is an example of one that exists but it not accessible to user processes.
    - If the PTE_U flag is present, the page is accessible to processes as well as the kernel. [The U stands for "user" and the user and the kernel are opposites]
        - The value of that flag is 4
- Why does the entry stick these flags into it?
    - Because the bottom 3 hexadecimal digits of every entry are not used for the address, because the addresses of the physical pages are aligned -- they're always multiples of 4096
        - So we ignore them for purposes of address finding (but use them for flags/permissions)
        - (NB: Means last 3 digits of VA and PA are the same)
    - So the system is saving space. (And confusing you)
    - So because the flag earlier (0x0007) was 7, the PTE_P, PTE_W, and PTE_U are all on.
        - (Why do we have three hex spaces for this? Because there are other flags -- but don't care or ask about them)
- One more. We want to map VA 0x10002003 to PA 0x0000A003
    - Do we need a new PTP? No, because the PDI is the same (64)
    - Which PTP do we modify? 2.
    - What do we set it to? A.
    - What are some interesting things about this?

- ○ Let’s assume all other VAs in the system are 0.
- ○ How many other physical addresses are accessible through this page directory?
    - ■ Can access 2 other pages
- ○ Can fit an 8192 size character into this because of the way VM works and what makes it cool
- ○ Once the OS sets up this system, all the memory you write cares only about the VAs
- ○ So if we write stuff into VA, doesn’t matter that PAs are separate. From the POV of your program, it’s contiguous, due to VM.
    - ■ Contiguous VA plus noncontiguous PA is “awesome”
        - ● It avoids fragmentation
    - ■ As long as the kernel can find pages no matter where in physical memory, it can cobble together something for the VA

● Now let’s add another virtual address space


P.A.S
0x0000
0x1000
0x2000
0x8000
0x9000
Hi
P.D.P
P.T.P
P.D.P #2
#0
#1
#2
↳ physical page # → multiply by pagesize for actual page addr.
• (pagenum << 12) == address
V.A.S
0x2000
↳ 1 to 1 mapping

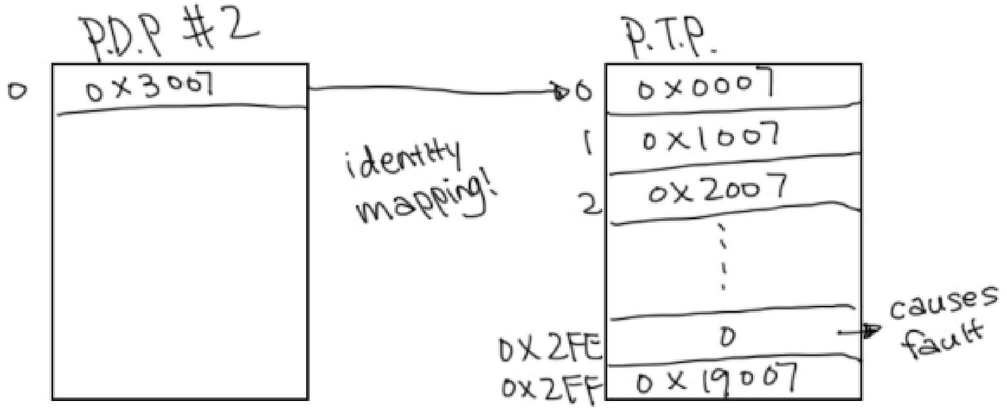
- ○ “BAAAADAAASSS” - Prof Kohler
- ○ Map 0x2000 to VA 0x2000
- ○ Where do we map to? 0
- ○ We need another PTP
- ○ Which entry do we modify and what do we need to put there?
- ○ Modify entry #2
- ○ Put 0x2000 there
- ○ Now there is a 1:1 mapping between 0x2000 and 0x2000



- Four more identity mappings:
  - At 1 can put 0x1007
  - At 0 can put 0x0007
  - At 4 can put 0x4007
  - At 3 can put 0x3007
- A problem! It seems as if pages that should be secret are now exposed
- For example, VA 0x3000 can be used to access a piece of memory used for a PTP
- And VA in VA space 2 can be used to access data that "belongs" to another process
- Whose problem is this? Processor's? OS'? Process'?
  - Problem for a processor looks like a fault
    - Gives us a fault when something is wrong
  - Can the processor handle this weird combo of page tables? Yes
  - Processor doesn't understand process isolation and stuff. It's stupid. It's happy with this page table.
  - If you access something that doesn't exist it will cause a fault, but this won't cause a fault.
- This process can access any part of physical memory. It can get control of any part of it.
- Suggestion: We want to modify the entry at VA space 2. Which VA contains it? To figure it out, can work backwards
- *(uint32_t *) 0x3008
  - This accesses VA space 2
- Set it to 0x10000
  - Is this enough? NO, because of flags
- 0x10007
- Now processes can modify their own page directories and thereby get access to everything at once

* Fork
  - Running the 3rd program
  - Runs the triangular numbers

- If we run it, how many lines of output do we expect?
    - 1001
- Got 127 instead... got an error message
- Problem caused by recursions. We've got to have a way to make recursive function that go down more than 1000
- Need more stack pages
    - 8-9 needed
- Why not allocate 9 to everyone?
  Reduces memory utilization
- If we assume we have other processes in the system, most of them don't need 9 full pages of stack
- We would be allocating pages that are unneeded
- How about we give the process stack only when it needs it?
- Could modify the compiler so that every time it called a function to check if it's at the top of the stac and if not explicitly allocates another stack page
    - Disadvantage: function calls become much more expensive
- So the solution is:
    - Instead of reporting an error message, allocate a page whenever we get a fault
- Stack page started at 0x0030000



    - And this zero causes the fault
- Previously we skipped instructions due to the fault; now we will allocate memory
- Allocate new page, then map it

- For extra credit:
    - Look up "copy on write" in the book
    - Otherwise, we'll go over it next Tuesday
    - Allows us to share memory between isolated processes
- VM is also very very dangerous
    - He put some assembly code in (asm("movl $0x40000, %%eax;
      movl %eax, %esp") under function app_printf in process_main);
    - It changes the stack pointer to point into the kernel space
    - Then it pagefaults because it can't write in the kernel space
    - But the kernel then allocates memory when that happens...

    - Causes the machine to reboot.
    - How do we fix this problem?
        - When giving a new page, check to see if the address is ge than the process space