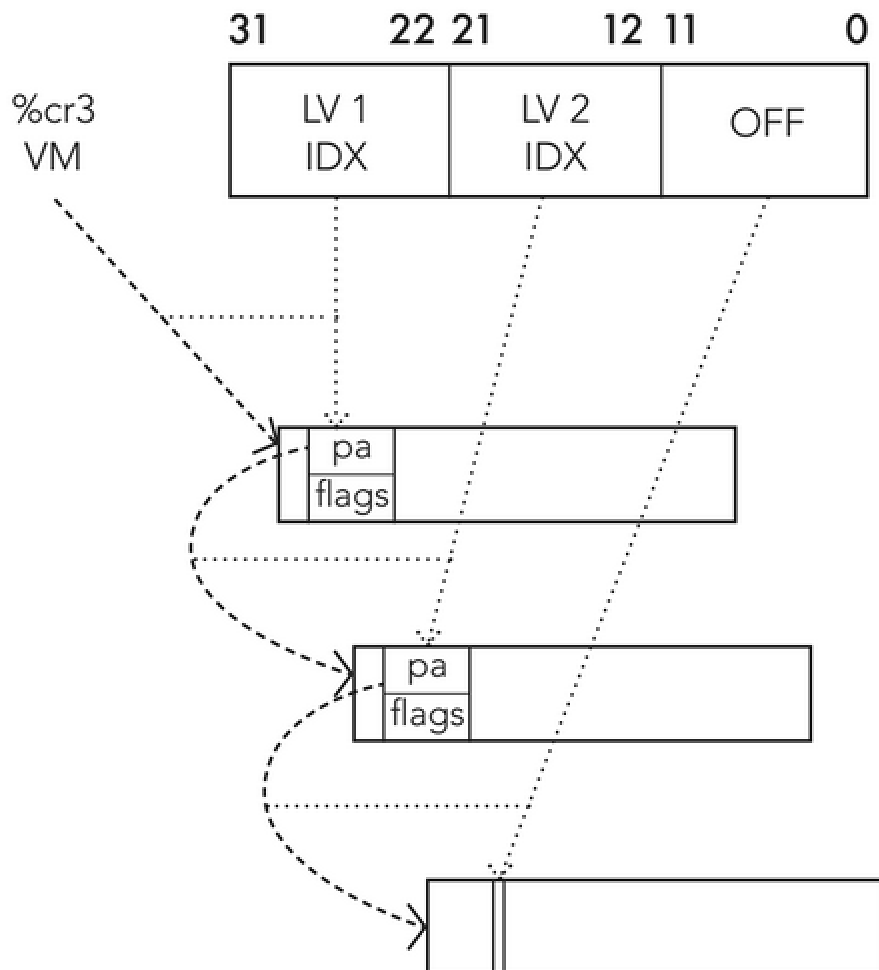
VIRTUAL MEMORY II

By Armaghan Behlum, Kyle Franseen, Linda Song, Jasmine Yan

Tips for homework:

- Put return values for exception in %eax
- Pid is process id
  - 1/process
  - Owner is pid most of the time.
  - Value for owner says that it belongs to the kernel
  - Another value says that it is reserved
- Do we ever work with memory by its physical address?
  - A few. One such installs the virtual memory page table.

x86 Page Table Review



- L1 page table located at %cr3 in VM.

- L1 index in address of L1 pagetable: find physical address of L2 pagetable and some flags
- DON'T FORGET THE FLAGS! If entry treated as a pointer, make sure to remove the flags.

- Bottom 12 bits possibly flags
  - Make sure to mask off the bottom 12 bits (not just the bottom 3!).
  - If the PTE_P bit is 0, then the entry is empty.
- If an unprivileged process tries to reference an address without PTE_P bit set, they will hit a fault.
- If the kernel tries --> page fault, since it also cannot reference a non-present address.
- If flag off in one level, it is treated as off on the other levels too.
  - **Ex**: if you have PTE_P bit set on L1 pagetable of the address, but not L2 pagetable, treated as unset.
  - Flags that matter are bitwise AND of all the levels.
- If PTE_U and PTE_W only set on L1, then unprivileged addresses will cause a fault if trying to reference the address.
  - However, the kernel would not when trying to read.
- When is it good that the kernel cannot write to some memory?
  - Same reason why you might declare constants in C.
- PTE_P - present. PTE_W - writeable. PTE_U - privileged.

OS 01

- In an ideal world, hello and welcome can keep swapping with each other. We saw a few attacks in the last few weeks.
  - Check out the v05 branch to see how hello is trying to turn off interrupts.
- In p-hello.c: add these four lines of code right after the "HA HA HA DIE DIE."

```
uint8_t* code_ptr = (uint8_t*) 0x40048;
code_ptr[0] = 0xeb;
code_ptr[1] = 0xfe;
sys_getpid();
```

- 0x00040048 is a kernel address for sys49_int_handler, the interrupt handler.
- The evil hello has put two instructions there from its own code!
  - The instruction causes it to just jump to itself, so it enters an infinite loop.
  - **Reason**: fe is -2, so it goes back twice to where eb is, and eb is a jump.
  - The jump is measured relative to the next instruction pointer. So jump 2 spots before the next instruction pointer, which is back to the eb, or the jump.
- OK for process to enter infinite loop since we have timer interrupts to handle them.
- But in this tiny OS, bad when kernel enters an infinite loop like this, since interrupts are always disabled for the kernel.
- In real OS, some parts where interrupts are disabled, i.e. startup when saying states.
- **Solution**: make sure processes cannot write into kernel memory (step 1 of pset!)
- This tiny OS also has a virtual_memory_map function which you can call with proper flags:

```
virtual_memory_map(kernel_pagetable, 0, 0, PROC_START_ADDR,
PTE_P | PTE_W);
```

- But causes an unexpected interrupt 14 = page fault. We need to re-map the console!

Console

- Physical portion of memory that can be used to talk to other devices.
- Array of memory mapping to chars on screen.
    - First 80 elements are top row of the screen.

Back to OS 1

- How can I make the console accessible? Map the console itself! Like so:
  ```
  virtual_memory_map(kernel_pagetable, console, console, PAGESIZE,
  PTE_P | PTE_W | PTE_U);
  ```
- How to fix the kernel accessing memory that is illegal? Jump to **v08**.
    - This process does something it should not, causing interrupt -> marks process as dead
- %cr2 contains the address that faulted when a page fault occurs.
- This system fault is a **trap** because the fault was intentionally called.
    - The instruction pointer is set to the next instruction when a trap occurs.
- What about a fault?
    - Fault time instruction pointer: instruction that failed, vs next instruction.
    - This lets us re-try the failed instruction.
- One solution for a problematic instruction is ignore it.
    - In this instance, works to jump 7 bytes forward (not all bad instructions will be 7 bytes!).
- Now jump to **v10**.
    - Similar to step 2 of pset
    - Hello is modifying the memory of another process!
    - One fix: make a new pagetable for hello
        - Not allowed to access another process's pagetable
        - Process 1 will pagefault, process 2 can continue running, but slowly
        - Reason: process was shifted to an infinite loop (because of earlier fix of add 7 trick).

Confused Deputy Problem

- Privileged code acts on behalf of unprivileged code.
    - Ex: Any system call
    - Problem: if privileged code is tricked into doing something inappropriate.
    - No combination of system calls should mess up process isolation!
- Our operation system has a ramdisk.
    - Allows user processes to use portion of memory like a disk
        - Can be read from or written to by the process.

    - Attempt by kernel: error checking so users don’t try to read/write more than the ramdisk allows
    - Also fails on integer overflows.
- See **v12** for confused deputy
    - Ramdisk lets us convince kernel for permission to run CLI
    - How? Writing to ramdisk permissions needed to run CLI
    - Then have ramdisk read to a buffer, where the registers actually are for process
        - Ramdisk will change the permissions of the process, so it can run CLI.
        - We’ve convinced kernel to do something the unprivileged process could not do!
    - To fix this, the kernel has to check if the user has permissions to write to that address.
- Another example: **os02/p-recurse**
    - Run the OS, then hit “b.” Doing so causes lots of function calls! Fault!
    - Why? Lots of recursive calls, eventually run out of stack space --> page fault.
        - Now in unmapped memory
    - **Solution**: allocate a page where there is unmapped memory. It works!