CS 61: Systems Programming
Scribe Notes: 10/31/2013

Announcements:
Bomb server is crashing currently, hopefully will be better soon
Guest lecturer next week talking about virtual memory stuff and what it is like to do systems research
If you update your lectures repository, there are now branches! (B01, B02, B03). Branches file in os01 describes the contents of the branches. The branches contain the code we've been stepping through in class. E.g. "git checkout origin/v02"

Last time:
Process Isolation:

- Process isolation requires that the kernel shares the system's resources (CPU time, memory, disk, screen, other hardware) fairly between all processes.
- 1st attack:
  - Attacker is trying to enter an infinite loop to monopolize one of those resources: CPU time
  - Kernel programmer introduced timer interrupts to prevent CPU monopolization
- 2nd attack:
  - Attacker runs cli instruction to prevent interrupts from happening; this is an attack on the interrupt hardware
  - Kernel programmer responds by reducing the process's privileges so the process cannot access interrupt hardware.
- 3rd attack:
  - Attacker makes an inappropriate modification on the kernel's memory
  - Kernel programmer responds by introducing virtual memory to protect the kernel's memory
- In each cycle, the attacker makes an attack on one of the system's resources, and the kernel responds by protecting that memory
  - Example: in mac OS 9, there was no protection of memory, which was bad.

Privilege:

- An abstraction used by the hardware to differentiate between safe and dangerous hardware operations. Dangerous hardware operations can only be executed by privileged code.
- Information on whether code is privileged is stored in a special register that requires privilege to change
  - On x86, privilege is stored on the lower two bits of the register %cs (aside, %gs often contains canary values)
    - 0 – kernel privilege (as much privilege as the system is willing to give)
    - 1 – not used
    - 2 – not used
    - 3 – process privilege
- Challenges with privilege
  - How can the process ask the kernel to do something?
    - It can't just call a kernel function directly because the kernel doesn't want processes to just be able to call any kernel function (e.g. turning off interrupts)

        - So, the kernel installs an **exception vector**, which is an array of entry points for exceptional control transfers. (using the instruction lidt)
        - int $N activates element N of the exception vector, which corresponds to the kernel running a certain system call.
        - This switches to kernel privilege, and stores %esp, %eip, %cs on the kernel stack
    - How can the kernel drop privilege and run a process?
        - The kernel needs to register a stack with the processor which stores the values of registers when the interrupt is called including the value in %cs and the return address that the kernel should return to when it is done with the system call.
        - The processor changes privilege when the interrupt is called
        - The kernel returns to the process by calling the iret instruction, which restores %esp, %eip, %cs from the kernel stack.
    - Note that this whole process is very expensive – this is one reason why system calls are so expensive

Traps vs. Interrupts vs. Faults:

- Trap – system call (interrupt the process wanted to take)
    - Eip is the instruction directly after
- Interrupt – something the hardware does
    - Eip is the instruction directly after
- Fault – something caused by an error the program made
    - Whenever a process tries to violate its privilege or hardware does something weird, the kernel needs to get control.
    - Eip is the problem instruction, **not** the next instruction. This tells you what the problematic instruction is. It also allows the kernel to try to fix the problem and then restart the program at the faulting instruction (or just skip the problematic instruction in the process and continue running). This is what happens with memory mapped io – page faults.
- The name of a fault that occurs at a virtual memory location is called a page fault

Virtual Memory:

- One of the most powerful tools at the hardware/software boundary
- Goal: protect kernel memory
    - On the processor level, we need to be able to mark memory as being accessible to only the kernel or to both the kernel and the process
- We want a compact representation of these memory regions
- To do this, we restrict the size and the offset of the memory region to be a multiple of pagesize, where pagesize is 4096 bytes – these are **pages.**
- X86 implements a 2 level page table to keep track of pages
    - Lower 12 bits of each address are a page offset
    - Middle 10 bits are indexes are the level 2 page index
    - Top 10 bits are the level 1 page index
    - To check an address, the hardware divides an address into these three types, throws away the page offset, then looks up the other two regions in the page table (which has address cr3 on x86)

        - The page table is an array of page entries – we use the level 1 index to look up the entry in this page table. If the value there is 0, the address is invalid, if the value is not 0, it points to a level 2 page index
        - Then we use the level 2 page index to look up the entry in the level 2 page table: if the value is 0, then the memory is invalid
- Other architectures can use different ways of keeping track of pages
- WeensyOS has addresses from 0x00000000 to 0x002FFFFF, so it needs a level 1 page table and only one level 2 page table (because the first 10 bits of every address is 0)
- In Linux, there's generally vast amounts of empty space between before a program's global data and between the globals and the stack, so the L1 page table only needs a couple of entries and we only need a few L2 page tables.