October 29 Lecture 16

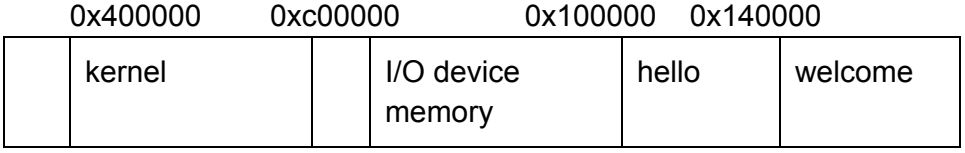
Sierra Katow, Raymond Kim

Process Isolation

- **Process:** An application program in execution
- **Isolation:** Each process believes it has full control of the processor
- Processes interact only at explicit interfaces (e.g. file system)
- 1 process can't touch another process's memory
- This allows a computer to still function even if one of its components fails
    - allows for failure isolation: if one process does something wrong, other processes can continue to work without being affected
    - if you have a serious failure or enough failing components, then the system as a whole will probably stop working

os01

- virtual machine provides illusion of hardware
- gdb can debug boot process of an emulated computer
- in os01, the kernel lives at low addresses (not normal)



- kernel allows multiple processes to run at same time
- when it jumps to 0x140000 then *hello* is yielding control to *welcome* (sys_yield)
- the processes = hello & welcome
- hello cannot directly start running welcome (it has to *allow* welcome to start running) since it can only interact at explicit interfaces
- system call: function call made by a process to kernel
    - schedule() is not a system call; it's a function within kernel
- exception control flow
    - 3 types
        1. **trap**: the one system call corr to. Exception control flow initiated on purpose.
            a. x86: "int" instruction
- normal control flow
    - process (hello) starts with normal control flow then runs into the exceptional control flow which causes the processor to put the process on hold and start running the kernel
- there needs to be some sort of timeout so a process can't run forever and the kernel can regain control
    - the process has an illusion of full control, but it doesn't actually have full control of

the processor
    - (just like os01 believes it is running on real hardware, but it's actually running on an emulator, running on an emulator, running on real hardware)
- the kernel gets control
    - interrupt: exceptional control flow initiated by hardware
    - if we had a piece of hardware that sent an interrupt every n seconds, it would hopefully solve the issue
    - "int" stands for interrupt
- timer_init() is a tiny driver that initiates a piece of hardware to interrupt a certain number of times a second
    - stored in khardware.c
    - "Panic! Unexpected interrupt!" ← shows that interrupt actually happened, but we don't know what to do when interrupt happens
    - interrupt happens, interrupted the kernel, so the kernel stops and outputs that message (b/c we haven't yet told the kernel what to do with the interrupt)
- **driver:** piece of software that handles hardware whose job is to interact with a piece of hardware
- **schedule** allows another process to run
    - it's not a call that processes are able to call directly (not a system call)
    - it's used in the implementation of other system calls
    - current is a var inside the kernel
    - (kernels are really just code that manipulate processes)
- When speeding up interrupts, why didn't the code speed up?
    - has to do with randomness in when the timer goes off (so if it interrupts welcome then it's wasted)
- fork bomb
    - what if we created a bunch of processes with infinite loops
    - but if os has timer interrupt, even though all those processes exist, welcome will still be able to run
- Can you change the schedule? Potentially, but it's in the kernel.
- So...we'll try to disable interrupts.
    - **cli()** disables interrupt
    - timer won't go off anymore and we can run infinite loop effectively
    - clearing interrupts is important for kernel to be able to do b/c think about if it was trying to configure hardware and then it got interrupted, then the hardware would be ½ configured
- kernel is dormant until activated (e.g. sys_yield)
    - activated by sys_yield (trap), timer interrupt
    - modify processor so it restricts cli so only the kernel can run it
- instructions can be either SAFE or DANGEROUS
    - **safe**: never violate process isolation
    - **dangerous**: can violate process isolation
        - dangerous instructions should only be executed by the kernel.

        - e.g. it's dangerous to change the lower 2 bits of the register
        - the hardware guys need to work in combination with os engineers to figure out which instructions might violate process isolation. hardware guys restrict those instructions so they can only be executed by kernel.
        - there are a set of registers which define whether the current process is the kernel or not
- **general protection fault:** occurs when process is confused
    - e.g. when process tries to do something it doesn't have permission to do, processor changes that to another type of exceptional control flow called exception
- **exception**: exceptional control flow initiated by software error
    - if this occurs, tell kernel that that process shouldn't run anymore to preserve process isolation
    - it works by using an array of pointers in memory which point to functions, one per interrupt number
    - when exceptional control flow is initiated, the processor looks up the right slot in that array and changes control flow to continue with whatever address it loads from the array
- what if we made storing things into memory a dangerous process?
    - this wouldn't be acceptable because memory is what programs are all about
    - we instead need to put some protection on memory itself
    - protection is called "virtual memory" (which is awesome)
- virtual memory map
    - can change permissions on a range of memory addresses to protect from memory attacks
    - virtual memory is manipulated by a function called **virtual_memory_map**
        - takes 5 arguments
        - permissions are the last argument
- three permission flags for virtual memory
    1. PTE_P: exists
    2. PTE_W: writable
    3. OTE_U: accessible to all user processes
- Right now all memory is set to being accessible by all processes
    - to change this, we take kernel portion of memory (starts at 0x0, goes to 0x100000) - then get rid of U flag
    - but user processes should be able to change the screen
        - so add another instance of virtual_memory_map for screen
- Does each process have its own page table? Not right now...that's another problem we'll fix in the future.