Scribe notes 10/29

OPERATING SYSTEM UNIT
How operating systems together provide PROCESS ISOLATION

Process Isolation:
What is a Process?

- An application program in execution
- Operating system deals with many processes at once
- As operating systems have gotten more interesting, the number of processes that you didn't start has been increasing drastically

Isolation:

- Each process believes that it has full control of the machine (processor)
- Processes can interact only at explicit interfaces (ex. File system)

What is good about it?

- One process can't touch another process's memory
- One component can fail and system can continue working
- You don't want the entire system to fail any time one element fails

os01 and problem set 4 have a similar code base:
No files in the operating system (cannot use stdio.h)
Commands:
"q" – kill
"make run" – start up virtual machine
"make run gdb" – run os01 as a virtual machine inside of the virtual machine: uses gdb to attach to the virtual machine
breakpoint at process_main (get through boot up)
continue: will stop at process_main in p-hello.c
x86 boot process
"sys_yield" yields control to welcome (only one processor, so sys_yield transfers the control of the processors)

- executes "int" (a system call)
- hello cannot directly start welcome
- only the kernel can do that

os01 memory layout:
Normal operating systems, kernel is at high addresses
In os01, kernel begins at 0x40000
Next comes I/O device memory (0xc00000- 0x100000)
Next comes process memory (divided up by process) (boundary between hello and welcome.c is 0x140000) (each of which have their own data, heap, stack)

System call:

- Function call by a process to the kernel

- Requires EXCEPTIONAL CONTROL FLOW (book chapter with this material)
- Trap: exceptional control flow initiated on purpose (in x86, this is the “int” instruction: “int” == “interrupt”)
- Put process on hold and start running the Kernel
- Control flow is “what instruction is being executed next”
- EXPECT: Kernel executes instructions and the will resume the hello program where it left off
- Yield system call for this os runs another process

How to avoid the process from infinite loop
- Only run 1 instruction from each process at a time
    - If we
- Timeout on thread
    - Process can enter an infinite loop but the processor can still maintain control
    - It is impossible to differentiate between very long processes and infinite process
    - But the goal is not to kill the process, as long as we give time to other processes too
    - Another type of exceptional control flow (INTERRUPT)
        - Exceptional control flow initiated by hardware
        - Send an interrupt every n seconds, giving control to the kernel

“timer_init” another tiny driver that initiates a piece of hardware that sends an interrupt every x times per second
We have to tell the kernel what to do once it receives an interrupt (add a new case for that interrupt)
When we receive that interrupt, run schedule() (aka, allow another process to run)
“schedule()” is a function within the kernel

Driver is a piece of software that interacts with hardware

In order for the process to reattack, you can disable the instruction calls:“cli()”
Processor must prevent processes from calling the cli instruction
- Restrict cli so that only the kernel can run it
-

Instructions can be SAFE or DANGEROUS:
SAFE: never violate process isolation
DANGEROUS: can violate process isolation

Processor takes DANGEROUS instructions and restricts them to the kernel memory (via hardware)

There are a set of registers that define whether or not you are in the kernel

The processor execute the instructions, NOT the Kernel. The kernel's instructions are waiting in memory.

Exception: exceptional control flow initiated by a software error (an "assert" is also an exception, but in pset 4, it stops the whole machine)

Interrupt 13: general protection fault (aka, the processor doesn't know what to do)
If a process faults (with a general protection fault), kill it

Process counterattack again:
Get the address of the system call and modify the kernel code to execute the infinite loop. The timer interrupt never interrupts the kernel (or at least part of it).

Processor counterattack: make the kernel's memory protected (cannot just make the memory calls dangerous because then functions would not be able to story any data)

"virtual_memory_map(pagetable, virtual address va, physical address pa, size sz, and permission perm)"

Permission flags:
PTE_P- memory exists
PTE_W- writable
PTE_U- accessible to all processes

Protect kernel virtual memory:
Virtual_memory_map(kernel_pagetable, 0, 0 0x10000, PTE_P|PTE_U)