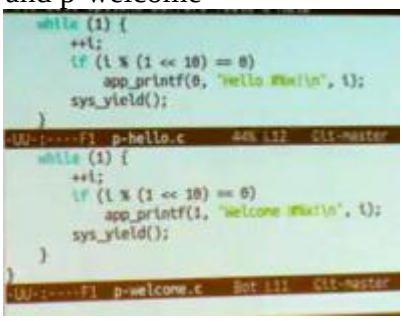
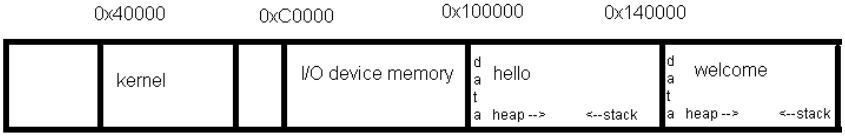
Oct 29, 2013

1. Process Isolation
    a. **Process**: An application program in execution
        i. The OS manages many processes at once.
            1. pstree : shows you all the processes running on your machine
    b. **Isolation**: each process believes it has full control of the machine of the processor
        i. Processes can interact only at explicit interfaces.
            1. Example: file system
        ii. Why is this great?
            1. Failure isolation
                a. If once process does something wrong, other processes can continue to work without being effected
2. os01
    a. 2 processes
        i. p-hello and p-welcome
            1. 

                a. loop to print Hello or Welcome repeatedly
                b. When you are working in an OS kernel, all the library calls are gone
                    i. No files
                c. Sys_yield allows the other process to run
                    i. So these processes are trading off
    b. Make run-gdb
        i. Starts up a virtual machine and uses GDB to attach to the virtual machine
            1. you can debug the boot process of an emulated computer
            2. you can step through the instructions the computer runs
        ii. set breakpoint at process_main, the opening part of our processes and step from there
            1. sys_yield compiles to “int.” with a value of 32. It’s behavior looks like a jump.
                a. you can see the program counter go from ~0x10021 to 0x40050

b. A picture of the memory:

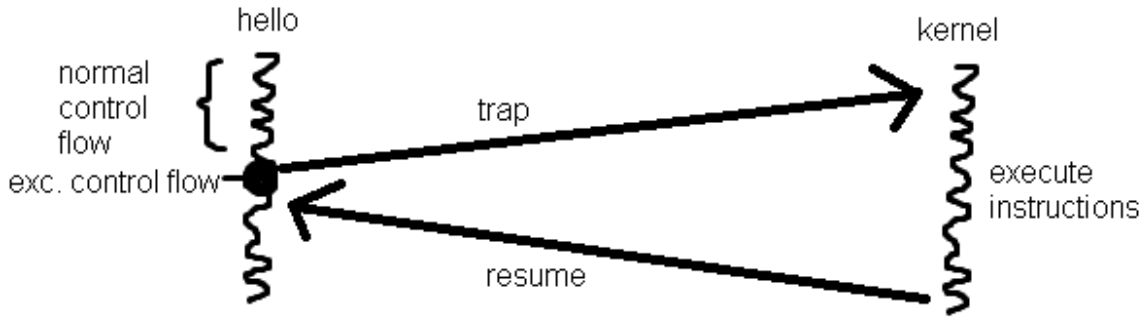


i. In normal OS's the kernel lives in high addresses, but in OS01 it its linked with a low address (0x40000)
ii. Then there is a neighborhood that consists of IO device memory (0xc0000 to 0x100000)
iii. Hello executes
    1. has data, heap, and stack
iv. Welcome executes.
    1. the boundary between them is at 0x140000
    2. has data, heap and stack

c. So it looks like an instruction in hello (int *0x32) handed over control back to the kernel

2. So the program jumps to welcome
    a. This is because only one processor can run at a time. Hello and welcome are isolated processes. So Hello cannot directly start running Welcome. It has to *allow* welcome to run. Only the kernel coordinate the action of multiple processes.
    b. That int instruction is making a **system call**.
        i. **System call**: function call mode by a process to kernel.
            1. since the process is acting like it owns the entire processor, transferring control from the process to the kernel requires **exceptional control flow**
            2. There are 3 types of exceptional control flow
                a. **Trap**: exception control flow initiated on purpose
                    i. Inth x86 this is done by the int instruction

c. Logically what is happening:



i. You have a process like hello, which is executing instructions like normal control flow.

ii. Then at some point the exceptional control flow is called.

1. Put process on hold and start running the kernel!
2. So the kernel gets control
   - **a.** you expect the kernel to execute instructions for the process
3. Then, the kernel will resume the hello program, exactly when it gets off

3. Evil processes….processes that try to do things that they should not be able to get away with if process isolation is correct

a. Hello is trying to enter an infinite loop, and the kernel is trying to avoid it. Hello should be allowed to enter an infinite loop itself, but Welcome should still be able to run.

i.

```
while (1) {
    ++i;
    if (i % (1 << 10) == 0) {
        app_printf(0, "Hello %x!\n", i);
        if (i % 4096 == 0) {
            app_printf(0, "HA HA HA HA DIE!\n");
        spinloop: goto spinloop;
        }
    }
    sys_yield();
}
```

ii. using gdb, you can see exactly where you end up in an infinite loop

1. there is an instruction that jumps to itself

iii. How can you fix this?

1. if a process only executes one at a time and in sequence, there is no way to prevent this problem. So we need support from the processor to prevent this attack.
2. we need some way to call timeout on a processor.
   - **a.** If a processor enters an infinite loop, then there is still a way for the kernel to regain control.
   - **b.** *The Halting Problem*: it is impossible to write a program that will determine whether another program will complete for every input
   - **c.** The goal of process isolation is to ensure that no one process monopolizes the machine. You can let

the infinite loop continue, as long as you give Welcome some time.

3. **interrupt**: exceptional control flow initiated by hardware
4. **driver:** piece of software whose job is to interact with hardware

iv. Updating the code

1. timer_init(1000);
    a. initiate a piece of hardware that will send an interrupt every 1000^{th} of a second
2. handle the interrupt
    a. case INT_TIMER:
    schedule(); // allows another process to run

v. How does the scheduler work?

1. current is a variable that remembers what the currently running process is. It loops over all the possible processes and runs the first one it encounters that is in the state 'runnable'

b. Now the kernel has a way to overcome the infinite loop. How do we get around the kernel? Disable interrupts by adding the following line of code: **cli()**;

i. The definition of the cli function:

1. static inline void cli(void) {
asn volatile("cli");}
2. the processor has given us an instruction that disables interrupts

ii. As a kernel, need to figure out a way to prevent the process from executing this function. A process really doesn't need the ability to use interrupts…

1. Modify processor so it restricts cli such that only the kernel can run it

iii. **Instructions can be either safe or dangerous.**

1. **Safe instructions** never violate process isolation
2. **Dangerous instructions** can violate process isolation
    a. Dangerous instructions should only be executed by the kernel.

iv. In our function call for starting the first process, get rid of the flag that gave processes permission to use cli. (Flag was only added for purpose of pedagogy. No normal kernel would do this…)

1. As soon as it tried to execute the cli function, it turned into an interrupt (13).
    a. Interrupt 13 is a general protection fault (what the hardware does when it is confused)
    b. Why is this fault happening?
        i. Process trying to execute function it doesn't have the rights to do. The processor change

that into another type of exceptional control flow, an **exception**

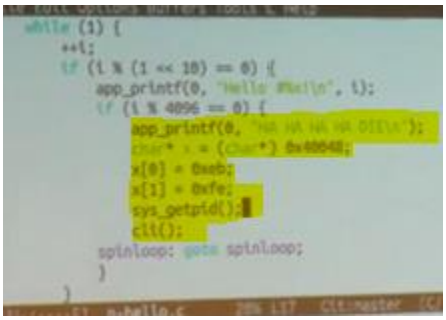
1. Exceptional control flow initiated by software error

v. When a process faults, we want to kill it

1. case INT_GPF:
   current->p_state = P_BLOCKED;
   break;

c. We know we can make a system call to start code running in kernel mode. Where are the system calls located? 0x40048 is the address for sys49

i. In hello:



1. Sys_getpid() has the system call number 49. This corresponds to the interrupt handler whose address we just changed.
2. This causes us to be stuck in the infinite loop.
3. sys49_int_handler jumps to itself *in kernel mode*
   a. The processor can be interrupted. The timer interrupt does not interrupt the kernel. So if a process can start that code and put what it wants there, that process has full control over the machine.

ii. Make the kernel memory inaccessible to processes

1. Programs need to be able to access memory, so we can't make memory dangerous. We need to put a layer of protection on memory itself: **virtual memory**
2. Virtual memory is manipulated by a function called *virtual_memory_map*
   a. It has 5 arguments: page table, virtual address, physical address, size, permission
   b. It changes the permissions on a range of addresses given by the virtual and physical addresses
3. When we initialized the virtual memory initially we made all of memory accessible to user processes
   a. Flags:
      i. **PTE_P**: the memory exists
      ii. **PTE_W**: the memory is write-able
      iii. **PTE_U**: the memory is accessible by processes

4. We can change this. Take the kernel portion of memory (from 0 to 0x100000) and make it so that user processes cannot access it:
    a. Virtual_memory_map(kernel_pagetable, 0, 0, 0x100000, PTE_P|PTE_W);

       // exception: still want user processes to be able to change the screen!
       Virtual_memory_map(kernel_pagetable, (uintptr_t) console, (uintptr_t) console, PAGESIZE, PTE_P|PTE_W|PTE_U);
5. At this point, still get an unexpected interrupt, meaning the process tried to write to memory it didn't have access to. Will fix that next time.

4. Preview for next time: The way that exception control flow works is there is an array of ptrs in memory, which point to functions, one per interrupt number. These handlers are the values stored into that stable. When e.c.f. is initiated, the processor looks up the right slot in that array, and changes control flow to continue at that address it loaded from the array
5. Review: Types of exceptional control flow
    a. **Interrupt**: e.c.f. initiated by hardware
    b. **Exception**: e.c.f. initiated by software error