Process isolation; Multiprocessing

One of the biggest ideas in computer systems → idea of a process.

OS from end of last class, 2 effective programs running, programs are running on virtual machine.

Virtual Machine
● Software implementation of a computer

What’s going on when you run VM?

Story time
CPU has dream of another CPU, that is connected to a whole bunch of hardware including a screen. When dream CPU performs instructions, it gets translated out of the hardware realm, and back into the instructions that make our virtual screen appear on our real screen.

hello.c

```c
    uint16_t *console = (uint16_t*) 0xB80000;
    *console = 0x0C00 | ‘@’;
    while (1)
        ;
```

● Just writing to memory...
● When run, see upper left corner of screen → @
● “0x0C00” → color

Two main interfaces with which CPU’s speak to hardware devices...

CPU’s and Hardware
● Programmed I/O
  ○ special instructions to interface with devices
  ○ Ex. inb, inw, outb, outl
● Memory mapped I/O
  ○ region of memory is used to interact with a device

What region of memory is being used to interface with the VGA console?

We are looking at an emulated VGA console.

What memory address does the VGA console start at? 0xB80000
Example: VGA console on x86 hardware is mapped as an array of 16-bit ints @0xb80000
laptop, there is a graphics card which watches a particular region of memory. As CPU puts data into memory, the graphics card reinterprets that data as an image and shows it on the screen. Many other video modes...but the same principle applies.

How does this VM show this image on the screen?
- Inside VM is a full OS, a real OS

What's going on inside VM?
- Memory like russian dolls

QEMU
- Borrowed memory from laptop, like programs borrow memory from computer
Who makes the connection between the memory and what is shown on the screen? Hardware.

In VM, what makes the connection between virtual piece of memory and the display? Software.

QEMU VM is a piece of software that behaves like hardware.

Instruction is bits + context
- OxEB OxFE = L2: jmp L2
- You can write a piece of software that says:
  ```c
  char *pc= ;
  if (pc[0] == 0xEB && pc[1] == OxFE)
    infinite loop;
  ```

The representation of programs as data, as memory, is the important key thing that allows us to make VM's.

Stored program computers (store instructions in memory) allow VM's.

Goal of this particular OS is printing “hello” and “welcome.” Two programs running. Introduce infinite loop into one process, the other process will never run. If we were to step through program...QEMU lets you debug entire computer using gdb-- it's just a program. Can run the entire piece of hardware, one instruction at a time.

Set breakpoint on “process_main”. First thing that process_main does-- reserves space for local variables. Step through one instruction at a time, characters gradually show up on the screen (HA HA HA HA). Each of those was simply writing a value into memory. QEMU VM interpreted that memory as a virtual VGA console then caused corresponding bits to be place into laptop’s display memory.

```
process.main () {
    while(1) {
        print ____;
        sys.yield();
    }
```

What is the purpose of sys.yield()? Sys.yield allows other programs to run, implements **cooperative multitasking**.

multitasking → more than one process ("hello" process, "welcome"
cooperative → processes voluntarily give up CPU

A lot of advantages → efficient
Giant disadvantage → vulnerable to attacks like an infinite loop

Alternative to cooperative multitasking is **preemptive multitasking**.

preemptive multitasking → a process can be forced to give up the CPU involuntarily... solves infinite loop attack

How can we force a process to give up a CPU? Need special features from CPU. We need interrupts.

**Interrupts & Exceptional control flow**
- Interrupt (exception) is an involuntary control transfer.
- The CPU changes its program counter from one location in memory to another when the original location is not a jump.
- Very simple, very powerful idea!
- CPU jumps to a new PC (%eip) due to some "external event"
- external events
  - interrupts → hardware
  - traps → software
  - faults → software error (ex. accessing memory that doesn’t exist)

What is an example of an interrupt the control flow of a process?
- A ticking clock that automatically interrupts periodically... then no infinite loop will take total control of the machine. This is called a timer interrupt.

**Timer interrupt**
- timer_init
- Timer interrupt is hardware. In order to interact with hardware, we need to use program I/O instructions or memory mapped I/O.
- Natural for display to use memory mapped I/O.
- Only argument passed to timer is rate at which it should go off.
- We need to know when we get a timer interrupt.
- Do different things in function depending on what interrupt there is.
- Program is continually interrupted and kernel is taking control.
- Although timer interrupt is taking control, we are not going to the other process.
- We need to do a system yield.
- When we get a timer interrupt, we call schedule();
- schedule();
  - searched for a run-able process then runs that process
  - this is an array search of an array of processes
  - round robin (e.i. on process 1, go to process 2)
Run a version of the code without the infinite loop...
- timer interrupts are still happening
- not doing many timer interrupts relative to how long it normally takes to yield the CPU
- Normally...
  - hello (five instructions) then welcome (5 instructions)
  - good hello -- gives up CPU willing
  - bad hello -- infinite loop
  - when hello takes control...
    - gives up only when interrupt
  - bad hello
    - hello executing 1000000 instructions
    - welcome executing 5 instructions
    - hello executing 1000000 instructions

What is responsibility of kernel?
- kernel is code that turned on the time interrupts
- linux is an example of a kernel
- a kernel is providing access to hardware resources divided fairly among multiple processes
- job → enforce fair access to hardware resources
- defines fair → no process dominates
- kernel is successful if no process cannot violate fairness

What if hello disabled the timer interrupt?
- cli(); → turns off all interrupts
- processor gives instruction that turns on timer interrupt
- cli() is a dangerous instruction!
- every CPU has both safe and dangerous instructions

**Safe Instructions**
- cannot violate **process isolation** (fairness property-- one process cannot monopolize the CPU)

**Dangerous Instructions**
- can violate process isolation

Who should be able to execute dangerous instructions?  ONLY the kernel.

How does the CPU know whether a process is running or a kernel?  If it is a good CPU, they should be able to distinguish.  There are special flags in certain registers that determine kernel or process privilege.  A kernel that is not very smart will give full privilege to applications.

When a process tries to execute a dangerous instruction that is not allowed?  It changes that instruction into an interrupt.  Can't execute instruction because it is dangerous and this gives the kernel a chance to fix things up.