Kernel Programming:
- Process isolation, goal to make programs run fast and reliably
  - Processes should not affect others, unless there's a specific and allowed communication channel
  - Each process can act as if it is in charge of the machine’s resources
- Code located in os01 directory, best run on cs50 appliance
  - What happens when a computer turns on?
    - Type “make run”; two processes occur, one prints hello and other prints welcome, both also print a counter
    - Investigate the code: each program has a while loop, which at the end yields so that the other program can run
  - We can single step through the hardware and examine memory: type “make run-gdb”
    - Notice the line at the beginning “the target architecture is assumed to be i8086”, this is an early (1970s) 16-bit microprocessor chip designed by Intel
    - The first things printed are the boot process: the leap from hardware instructions into software.
  - Type “b boot_start”
    - Now running in code written by Eddie
    - Point is to bring machine up to date
    - Loads the kernel into memory
  - “b kernel
    - Initializes the hardware
    - clears the screen
- Understanding sys_yield();
  - Tells the system to allow other processes to run
  - Seemingly dangerous, gives system control to the program, which takes away from the Kernel’s privilege: only the Kernel can access hardware.
- Exceptional Control Transfer
  - Means of safely transferring control between privileged and unprivileged code, demonstrated below:

Stepping through a system call
- Our goal is to maintain the process' and kernel's appropriate privilege levels during the system call by preventing the process (unprivileged) from inappropriately having a higher privilege
- GDB is set up to give us a bunch of information on each stop:
  - The current line of C code, if appropriate
  - Next five instructions in assembly
- sys_yield() is translated to the following assembly instruction:

```plaintext
int $0x32 // call interrupt number 50
```
This one instruction performs these operations:

- Jumps the program counter into the kernel's code (lower memory addresses)
  - The particular address is not stored within any register of the current state of memory
  - Changes the current stack pointer (%esp) to a lower memory address
- Unconventional because both the instruction pointer and the stack pointer are significantly changed
- int stores the location of the next instruction within the process after the interrupt, the previous stack pointer, and other important process-specific registers on the kernel's stack
- Overall goal is that we need to transfer control from unprivileged code to privileged code in a protected way
  - This is accomplished by having the kernel designate a small number of specific entry points (function within the kernel that a process may call)
  - These are the only way for the process to access the kernel
  - There are 256 possible different interrupts, so there are 256 possible entry points into the kernel
  - This cannot be done in pure software, we need hardware to enforce this
- On X86, entry points are implemented as "software traps" using the int (interrupt) instruction

### Before the process runs

- Kernel sets up an interrupt table, which specifies what code to run when an interrupt is called (mapping between integer to kernel function addresses). See os01/k-exception.S
- This interrupt table is registered with the hardware

### When the process executes int

- Hardware looks up the entry within the interrupt table
- Finds address of the kernel function as well as the address of the kernel stack
  - We cannot use the process' stack because the process has access to changing it (could set to 0, which would cause a reboot)
- It's important that everything is under the kernel's control when executing a kernel function
- Saves the return address and other registers on the kernel stack
- Hardware raises its privilege
- This similar to a function call, but because the call is across privilege levels, the current state of the machine needs to be fully saved (not just the return address)
- Side note: Javascript X86 emulator (http://bellard.org/jslinux/)
  - This is Javascript pretending that it is an X86 machine
  - There is an interpreter that executes the instructions
- Is there a hardware privilege level that prevents programs from installing their own exception table? Yes!
  - Current privilege level - 0 when in kernel and 3 when in process
    - This is stored within a special register designated for systems programming called %cs -- specifically the lower two bits, previous value of which also needs to be stored on interrupts
    - If the privilege level is not 0, the hardware will prevent an override of the interrupt table
    - The processor will fault here and give the kernel control-- whenever the process tries to execute something outside of its privilege, the processor will hand control back to the kernel
- The processor prevents processes from directly overriding the lower two bits %cs register and just ignores any attempts to change them other than through the int instruction (or by seg. faulting, etc)
- How is privilege a hardware concept?
  - There's a set of "dangerous instructions" -- can only be executed in privileged mode
  - These instructions, when you look at the definition, are hard wired to perform different things based off of those lower two bits in %cs
  - Example: lidt which loads a new interrupt table
- When a kernel executes a process, it just gives the instruction pointer to the process so that the process is running directly on the machine
  - This means that the process could access invalid memory
  - However, there is no way for the kernel to check what memory the process is trying to access (otherwise, it would have to be an interpreter, which is a big cost to pay)
  - Instead, operating system designers and processor designers have developed a set of interfaces to enforce process isolation, even when unprivileged code has control over the processor
  - The processor notices the fault and then passes control to the kernel via an interrupt
- Why don’t we just call sys_yield() in our standard C programs?
  - The system automatically performs a timer interrupt - this means that the kernel will take control of the processor every so often
○ Example used in class is that this occurs every millisecond
○ Also, most of the time, modern computers are mostly idle even when lots of processes are running, so this isn’t necessary to expose to the programmer
● Aside: if cumu (sp?) is virtualizing a process, why is its usage changing?
○ What this is is a while loop - it looks at the current instruction pointer and loads data from an array rather than from a disk
○ Basically its an emulator that is simulating loading instructions rather than actually loading them
○ It is sufficiently fast, so is used, but isn’t the actual processor!

What happens after the interrupt?
● We call exception.c, of course!
  ○ Exception.c is the code that is called whenever an interrupt takes place
  ○ It only has one argument - a saved copy of all of the register states from the process that was interrupted, x86_registers.
  ○ Lets now look at what this object is.
● Aside: What is goin’ on with x86_registers?
  ○ See x_86.h - this just holds all of the registers that might matter to the program - you can find it here.
  ○ We need this since to recover where the program was, we need to keep track of every part of its state that was saved - hence, the registers
  ○ There is one exception - %eax, which we keep for a return value
  ○ For each process, there can be one of these.
● Then, we call a scheduler
  ○ A scheduler tells us what program to run next after an exception was called
  ○ It looks for a runnable process that isn’t the current one, then we run it and call iret (explained below)
  ○ Be careful here - if written badly, it is possible that it is in the scheduler that process isolation doesn’t occur
  ○ For example, if we just call run(current), which will run whatever thread was run last, then process isolation won’t occur at all - a piece of code can run forever
  ○ A problem is that we can’t even figure out the backtrace from the stack - we’ve moved to a different stack after all
● What is iret?
  ○ iret is the inverse function of int - it will take the processor and transfer it from privileged code to unprivileged code
  ○ It also conveniently loads all of the saved stack information to the registers. We then call run to start the program
● But - what is the speed of everything?
  ○ Interrupts: slower than calling return by 100s of times.
This is because the processor actually is constantly executing 5-20 instructions that are partially finished - interrupts have to “unwind” all of these so that we can move

- Call: slower, but not so bad
  - There’s no privilege change, so it doesn’t matter to the processor if it continues to execute the same bunch of instructions

### How can we break process isolation?

- **Infinite Loop**
  - If a program just has an infinite loop (consider if we ran `spinloop; goto spinloop;`), then it will never call `sys_yield()` automatically.
  - Remember the concept of timer interrupts from before? We turn on a timer that interrupts once per millisecond, and calls `sys_yield()` for us.

- **Bad Scheduler**
  - (See above) what if the scheduler called `run(current)` instead of looking for the next one?
  - The timer interrupt happens, but even this doesn’t save us - the problem is that we just go back to the same program whenever we have interrupted

- **Dangerous Functions**
  - Lets think about the function `cli()` - it stands for “clear interrupt flags”.
  - If we do this, then interrupts won’t happen, and the program will get to run forever.
  - However, if we call it usually, it’s actually not allowed to run. If we try to execute it, instead, we end up with a “general protection fault”.
    - Note: This is because our kernel isn’t sufficiently advanced in order to understand what to do.
    - The processor has told the kernel that a fault exists, but the kernel doesn’t know what to do so it just crashes to protect itself