```
6.828 2014 L5: Isolation mechanisms
==

* OS design driven by isolation, multiplexing, and sharing

* What is isolation
  the process is the unit of isolation
  prevent process X from wrecking or spying on process Y
    memory, cpu, FDs, resource exhaustion
  prevent a process from wrecking the operating system itself
    i.e. from preventing kernel from enforcing isolation
  in the face of bugs or malice
    e.g. a bad process may try to trick the h/w or kernel

* what are all the mechanisms that keep processes isolated?
  user/kernel mode flag
  address spaces
  timeslicing
  system call interface

* the foundation of xv6's isolation: user/kernel mode flag
  controls whether instructions can access privileged h/w
  called CPL on the x86, bottom two bits of %cs
    CPL=0 -- kernel mode -- privileged
    CPL=3 -- user mode -- no privilege
  x86 CPL protects everything relevant to isolation
    writes to %cs (to defend CPL)
    every memory read/write
    I/O port accesses
    control register accesses (eflags, %cs4, ...)
  every serious microprocessor has something similar

* user/kernel mode flag is not enough
  protects only against direct attacks on the hardware
  kernel must configure control regs, page tables, &c to protect other stuff
    e.g. kernel memory

* how to do a system call -- switching CPL
  Q: would this be an OK design for user programs to make a system call:
    set CPL=0
    jmp sys_open
    bad: user-specified instructions with CPL=0
  Q: how about a combined instruction that sets CPL=0,
    but *requires* an immediate jump to someplace in the kernel?
    bad: user might jump somewhere awkward in the kernel
  the x86 answer:
    there are only a few permissible kernel entry points
    INT instruction sets CPL=0 and jumps to an entry point
    but user code can't otherwise modify CPL or jump anywhere else in kernel
  system call return sets CPL=3 before returning to user code
    also a combined instruction (can't separately set CPL and jmp)
    but kernel is allowed to jump anywhere in user code

* the result: well-defined notion of user vs kernel
  either CPL=3 and executing user code
  or CPL=0 and executing from entry point in kernel code
  not:
    CPL=0 and executing user
    CPL=0 and executing anywhere in kernel the user pleases

* how to isolate process memory?
  idea: "address space"
  give each process some memory it can access
    for its code, variables, heap, stack
```

```
  prevent it from accessing other memory (kernel or other processes)

* how to create isolated address spaces?
  xv6 uses x86 "paging hardware"
  MMU translates (or "maps") every address issued by program
    VA -> PA
    instruction fetch, data load/store
    for kernel and user
    there's no way for any instruction to directly use a PA
  MMU array w/ entry for each 4k range of "virtual" address space
    refers to phy address for that "page"
    this is the page table
  o/s tells h/w to switch page table when switching process
  why isolated?
    each page table entry (PTE) has a bit saying if user-mode instructions can use
    kernel only sets the bit for the memory in current process's address space
  paging h/w used in many ways, not just isolation
    e.g. copy-on-write fork(), see Lab 4
  note: you don't need paging to isolate memory
    type safety, JVM, Singularity
    but paging is the most popular plan

how to isolate CPU?
  prevent a process from hogging the CPU, e.g. buggy infinite loop
  how to force uncooperative process to yield
  h/w provides a periodic "clock interrupt"
    forcefully suspends current process
    jumps into kernel
    which can switch to a different process
  kernel must save/restore process state (registers)
  totally transparent, even to cooperative processes
  called "pre-emptive context switch"
  note: traditional, but maybe not perfect; see exokernel paper

back to system calls
  i've talked a lot about how o/s isolates processes
  but need user/kernel to cooperate! user needs kernel services.
  what should user/kernel interaction look like?
  can't let user r/w kernel mem (well, you can, later...)
  kernel can r/w user mem
    but don't want to do this too much!
  so style of system call interface is pretty simple
    integers, strings (copying only), user-allocated buffers
    no objects, data structures, &c
    never any doubt about who owns memory

## Xv6 internal overview

* Trace the first process, and its first syscall
  We will see all mechanism in action

* Process overview (unit of isolation)
  Each process has its own address space
    also includes kernel
    implemented using virtual memory
  Each process has its own thread of execution
    user stack
        kernel stack
  See proc.h

* First address space
  Kernel starts with an initial address space (see entrypgdir)
    Maps 4Mbyte (1 entry entry using 4Mbyte page!)
        Type info pg once we enter main()
        Draw diagram
```

```
    Kernel creates a new address space in C code (setupkvm)
      Draw diagram
      0x80000000:0x80100000 -- map low 1MB devices (for kernel)
      0x80100000:?          -- kernel instructions/data
      ?         :0x8E000000 -- 224 MB of DRAM mapped here
      0xFE000000:0x00000000 -- more memory-mapped devices
     See kvmalloc()
      Type info pg after switchkvm() into qemu monitor

  * First process (see userinit)
    allocproc(): set up stack for "returning" to user space
      draw picture of initial stack
    Fill in kernel part of address space (setupkvm again)
    Fill in user part of address space
      1 page containing initcode (see initcode.S)
    Setup trapframe to exit kernel:
      User-mode bit
          Eip = 0
          User-stack lives at top of 1 page of initcode
    Set process to runnable

  * Running first process
    Scheduler selects a runnable process (i.e., initcode)
    switchuvm:
      change to process address space
          set up task segment with kernel stack
          enable interrupts
      running now within process address space!
          type info pg into qemu monitor
    single step through swtch
      proc->context was setup in allocproc()
          initialized through trapframe!
      "returns" to user space

  * Executing first system call
    initcode calls exec  (with what arguments?)
    int instruction enters kernel again
    vector.S, trapasm.S, trap.c
    syscall.c
```