6.828 2017 Lecture 1: O/S overview

Overview

* 6.828 goals
  * Understand operating system design and implementation
  * Hands-on experience by building small O/S

* What is the purpose of an O/S?
  * Support applications
  * Abstract the hardware for convenience and portability
  * Multiplex the hardware among multiple applications
  * Isolate applications in order to contain bugs
  * Allow sharing among applications
  * Provide high performance

* What is the O/S design approach?
  * the small view: a h/w management library
  * the big view: physical machine -> abstract one w/ better properties

* Organization: layered picture
   h/w: CPU, mem, disk, &c
   kernel services
   user applications: vi, gcc, &c
  * we care a lot about the interfaces and internal kernel structure

* What services does an O/S kernel typically provide?
  * processes
  * memory allocation
  * file contents
  * directories and file names
  * security
  * many others: users, IPC, network, time, terminals

* What does an O/S abstraction look like?
  * Applications see them only via system calls
  * Examples, from UNIX (e.g. Linux, OSX, FreeBSD):

```
fd = open("out", 1);
write(fd, "hello\n", 6);
pid = fork();
```

* Why is O/S design/implementation hard/interesting?
  * the environment is unforgiving: quirky h/w, weak debugger
  * it must be efficient (thus low-level?)
	...but abstract/portable (thus high-level?)
  * powerful (thus many features?)
	...but simple (thus a few composable building blocks?)
  * features interact: `fd = open(); ...; fork()`
  * behaviors interact: CPU priority vs memory allocator
  * open problems: security; performance

* You'll be glad you learned about operating systems if you...
  * want to work on the above problems
  * care about what's going on under the hood
  * have to build high-performance systems
  * need to diagnose bugs or security problems

Class structure

* See web site: https://pdos.csail.mit.edu/6.828

* Lectures
  * O/S ideas
  * detailed inspection of xv6, a traditional O/S
  * xv6 programming homework to motivate lectures
  * papers on some recent topics

* Labs: JOS, a small O/S for x86 in an exokernel style

```
  * you build it, 5 labs + final lab of your choice
  * kernel interface: expose hardware, but protect -- few abstractions!
  * unprivileged user-level library: fork, exec, pipe, ...
  * applications: file system, shell, ..
  * development environment: gcc, qemu
  * lab 1 is out

* Two exams: midterm during class meeting, final in finals week

Introduction to system calls

* 6.828 is largely about design and implementation of system call
interface. let's look at how programs use that interface.
we'll focus on UNIX (Linux, Mac, POSIX, &c).

* a simple example: what system calls does "ls" call?
  * Trace system calls:
    * On OSX: sudo dtruss /bin/ls
    * On Linux: strace /bin/ls
  * so many system calls!

* example: copy input to output
  cat copy.c
  cc -o copy copy.c
  ./copy
  read a line, then write a line
  note: written in C, the traditional O/S language

  * first read/write argument is a "file descriptor" (fd)
    passed to kernel to tell it what "open file" to read/write
    must previously have been opened, connects to file/device/socket/&c
    UNIX convention: fd 0 is "standard input", 1 is "standard output"

  * sudo dtruss ./copy
    read(0x0, "123\n\0", 0x80)           = 4 0
    write(0x1, "123\n@\213\002\0", 0x4)          = 4 0

* example: creating a file
  cat open.c
  cc -o open open.c
  ./open
  cat output.txt
  note: creat() turned into open()
  note: can see actual FD with dtruss
  note: this code ignores errors -- don't be this sloppy!

* example: redirecting standard output
  cat redirect.c
  cc -o redirect redirect.c
  ./redirect
  cat output.txt
  man dup2
  sudo dtruss ./redirect
  note: writes output.txt via fd 1
  note: stderr (standard error) is fd 2 -- that's why creat() yields FD 3

* a more interesting program: the Unix shell.
  * it's the Unix command-line user interface
  * it's a good illustration of the UNIX system call API
  * some example commands:
    ls
    ls > junk
    ls | wc -l
    ls | wc -l > junk
  * the shell is also a programming/scripting language
    cat > script
      echo one
      echo two
    sh < script
```

```
  * the shell uses system calls to set up redirection, pipes, waiting
    programs like wc are ignorant of input/output setup

* Let's look at source for a simple shell, sh.c

  * main()
    basic organization: parse into tree, then run
    main process: getcmd, fork, wait
    child process: parsecmd, runcmd
    why the fork()?
      we need a new process for the command
    what does fork() do?
      copies user memory
      copies kernel state e.g. file descriptors
      so "child" is almost identical to "parent"
      child has different "process ID"
      both processes now run, in parallel
      fork returns twice, once in parent, once in child
      fork returns child PID to parent
      fork returns 0 to child
      so sh calls runcmd() in the child process
    why the wait()?
    what if child exits before parent calls wait()?

  * runcmd()
    executes parse tree generated by parsecmd()
    distinct cmd types for simple command, redirection, pipe

  * runcmd() for simple command with arguments
    execvp(cmd, args)
    man execvp
    ls command &c exist as executable files, e.g. /bin/ls
    execvp loads executable file over memory of current process
    jumps to start of executable -- main()
    note: execvp doesn't return if all goes well
    note: execvp() only returns if it can't find the executable file
    note: it's the shell child that's replaced with execvp()
    note: the main shell process is still wait()ing for the child

  * how does runcmd() handle I/O redirection?
    e.g. echo hello > junk
    parsecmd() produces tree with two nodes
      cmd->type='>', cmd->file="junk", cmd->cmd=...
      cmd->type=' ', cmd->argv=["echo", "hello"]
    the open(); dup2() causes FD 1 to be replaced with FD to output file
    it's the shell child process that changes its FD 1
    execvp preserves the FD setup
    so echo runs with FD 1 connected to file junk
    again, very nice that echo is oblivious, just writes FD 1

  * why are fork and exec separate?
    perhaps wasteful that fork copies shell memory, only
      to have it thrown away by exec
    the point: the child gets a chance to change FD setup
      before calling exec
    and the parent's FD set is not disturbed
    you'll implement tricks to avoid fork() copy cost in the labs

  * how does the shell implement pipelines?
    $ ls | wc -l

  * the kernel provides a pipe abstraction
    int fds[2]
    pipe(fds)
    a pair of file descriptors: a write FD, and a read FD
    data written to the write FD appears on the read FD

  * example: pipe1.c
    read() blocks until data is available
```

  write() blocks if pipe buffer is full

* pipe file descriptors are inherited across fork
  so pipes can be used to communicate between processes
  example: pipe2.c
  for many programs, just like file I/O, so pipes work for stdin/stdout

* for ls | wc -l, shell must:
  - create a pipe
  - fork
  - set up fd 1 to be the pipe write FD
  - exec ls
  - set up wc's fd 0 to be pipe read FD
  - exec wc
  - wait for wc
  [diagram: sh parent, ls child, wc child, stdin/out for each]
  case '|' in sh.c
  note: sh close()es unused FDs
        so exit of writer produces EOF at reader

* you'll implement pieces of a shell in an upcoming homework

6.828 Lecture Notes: x86 and PC architecture

Outline

- PC architecture
- x86 instruction set
- gcc calling conventions
- PC emulation

PC architecture

- A full PC has:
  - an x86 CPU with registers, execution unit, and memory management
  - CPU chip pins include address and data signals
  - memory
  - disk
  - keyboard
  - display
  - other resources: BIOS ROM, clock, ...
- We will start with the original 16-bit 8086 CPU (1978)
- CPU runs instructions:

```
for(;;){
        run next instruction
}
```

- Needs work space: registers
  - four 16-bit data registers: AX, BX, CX, DX
  - each in two 8-bit halves, e.g. AH and AL
  - very fast, very few
- More work space: memory
  - CPU sends out address on address lines (wires, one bit per wire)
  - Data comes back on data lines
  - *or* data is written to data lines
- Add address registers: pointers into memory
  - SP - stack pointer
  - BP - frame base pointer
  - SI - source index
  - DI - destination index
- Instructions are in memory too!
  - IP - instruction pointer (PC on PDP-11, everything else)
  - increment after running each instruction
  - can be modified by CALL, RET, JMP, conditional jumps
- Want conditional jumps
  - FLAGS - various condition codes
    - whether last arithmetic operation overflowed
    - ... was positive/negative

        - ... was [not] zero
        - ... carry/borrow on add/subtract
        - ... etc.
        - whether interrupts are enabled
        - direction of data copy instructions
    - JP, JN, J[N]Z, J[N]C, J[N]O ...
- Still not interesting - need I/O to interact with outside world
    - Original PC architecture: use dedicated *I/O space*
        - Works same as memory accesses but set I/O signal
        - Only 1024 I/O addresses
        - Accessed with special instructions (IN, OUT)
        - Example: write a byte to line printer:

```
#define DATA_PORT    0x378
#define STATUS_PORT  0x379
#define   BUSY 0x80
#define CONTROL_PORT 0x37A
#define   STROBE 0x01
void
lpt_putc(int c)
{
  /* wait for printer to consume previous byte */
  while((inb(STATUS_PORT) & BUSY) == 0)
    ;

  /* put the byte on the parallel lines */
  outb(DATA_PORT, c);

  /* tell the printer to look at the data */
  outb(CONTROL_PORT, STROBE);
  outb(CONTROL_PORT, 0);
}
```

    - Memory-Mapped I/O
        - Use normal physical memory addresses
            - Gets around limited size of I/O address space
            - No need for special instructions
            - System controller routes to appropriate device
        - Works like ``magic'' memory:
            - *Addressed* and *accessed* like memory, but ...
            - ... does not *behave* like memory!
            - Reads and writes can have ``side effects''
            - Read results can change due to external events
- What if we want to use more than 2^16 bytes of memory?
    - 8086 has 20-bit physical addresses, can have 1 Meg RAM
    - the extra four bits usually come from a 16-bit "segment register":
    - CS - code segment, for fetches via IP
    - SS - stack segment, for load/store via SP and BP
    - DS - data segment, for load/store via other registers
    - ES - another data segment, destination for string operations
    - virtual to physical translation: pa = va + seg*16
    - e.g. set CS = 4096 to execute starting at 65536
    - tricky: can't use the 16-bit address of a stack variable as a pointer
    - a *far pointer* includes full segment:offset (16 + 16 bits)

  - tricky: pointer arithmetic and array indexing across segment boundaries
- But 8086's 16-bit addresses and data were still painfully small
  - 80386 added support for 32-bit data and addresses (1985)
  - boots in 16-bit mode, boot.S switches to 32-bit mode
  - registers are 32 bits wide, called EAX rather than AX
  - operands and addresses that were 16-bit became 32-bit in 32-bit mode, e.g. ADD does 32-bit arithmetic
  - prefixes 0x66/0x67 toggle between 16-bit and 32-bit operands and addresses: in 32-bit mode, MOVW is expressed as 0x66 MOVW
  - the .code32 in boot.S tells assembler to generate 0x66 for e.g. MOVW
  - 80386 also changed segments and added paged memory...
- Example instruction encoding

```
b8 cd ab                16-bit CPU,  AX <- 0xabcd
b8 34 12 cd ab          32-bit CPU, EAX <- 0xabcd1234
66 b8 cd ab             32-bit CPU,  AX <- 0xabcd
```

x86 Physical Memory Map

- The physical address space mostly looks like ordinary RAM
- Except some low-memory addresses actually refer to other things
- Writes to VGA memory appear on the screen
- Reset or power-on jumps to ROM at 0xfffffff0 (so must be ROM at top...)

```
+------------------+  <- 0xFFFFFFFF (4GB)
|      32-bit      |
|  memory mapped   |
|     devices      |
|                  |
/\/\/\/\/\/\/\/\/\/\

/\/\/\/\/\/\/\/\/\/\
|                  |
|      Unused      |
|                  |
+------------------+  <- depends on amount of RAM
|                  |
|                  |
| Extended Memory  |
|                  |
|                  |
+------------------+  <- 0x00100000 (1MB)
|     BIOS ROM     |
+------------------+  <- 0x000F0000 (960KB)
|  16-bit devices, |
|  expansion ROMs  |
+------------------+  <- 0x000C0000 (768KB)
|   VGA Display    |
+------------------+  <- 0x000A0000 (640KB)
|                  |
|    Low Memory    |
|                  |
+------------------+  <- 0x00000000
```

x86 Instruction Set

- Intel syntax: `op dst, src` (Intel manuals!)
- AT&T (gcc/gas) syntax: `op src, dst` (labs, xv6)

    - uses b, w, l suffix on instructions to specify size of operands
- Operands are registers, constant, memory via register, memory via constant
- Examples:

AT&T syntax	"C"-ish equivalent	
movl %eax, %edx	edx = eax;	*register mode*
movl $0x123, %edx	edx = 0x123;	*immediate*
movl 0x123, %edx	edx = *(int32_t*)0x123;	*direct*
movl (%ebx), %edx	edx = *(int32_t*)ebx;	*indirect*
movl 4(%ebx), %edx	edx = *(int32_t*)(ebx+4);	*displaced*

- Instruction classes
    - data movement: MOV, PUSH, POP, ...
    - arithmetic: TEST, SHL, ADD, AND, ...
    - i/o: IN, OUT, ...
    - control: JMP, JZ, JNZ, CALL, RET
    - string: REP MOVSB, ...
    - system: IRET, INT
- Intel architecture manual Volume 2 is *the* reference

gcc x86 calling conventions

- x86 dictates that stack grows down:

Example instruction	What it does
pushl %eax	subl $4, %esp movl %eax, (%esp)
popl %eax	movl (%esp), %eax addl $4, %esp
call 0x12345	pushl %eip (*) movl $0x12345, %eip (*)
ret	popl %eip (*)

  (*) *Not real instructions*
- GCC dictates how the stack is used. Contract between caller and callee on x86:
    - at entry to a function (i.e. just after call):
        - %eip points at first instruction of function
        - %esp+4 points at first argument
        - %esp points at return address
    - after ret instruction:
        - %eip contains return address
        - %esp points at arguments pushed by caller
        - called function may have trashed arguments
        - %eax (and %edx, if return type is 64-bit) contains return value (or trash if function is `void`)
        - %eax, %edx (above), and %ecx may be trashed
        - %ebp, %ebx, %esi, %edi must contain contents from time of `call`
    - Terminology:

        - %eax, %ecx, %edx are "caller save" registers
        - %ebp, %ebx, %esi, %edi are "callee save" registers
- Functions can do anything that doesn't violate contract. By convention, GCC does more:
    - each function has a stack frame marked by %ebp, %esp

```
               +------------+   |
               | arg 2      |   \
               +------------+    >- previous function's stack frame
               | arg 1      |   /
               +------------+   |
               | ret %eip   |   /
               +============+
               | saved %ebp |   \
        %ebp-> +------------+   |
               |            |   |
               |   local    |   \
               | variables, |    >- current function's stack frame
               |    etc.    |   /
               |            |   |
               |            |   |
        %esp-> +------------+   /
```

    - %esp can move to make stack frame bigger, smaller
    - %ebp points at saved %ebp from previous function, chain to walk stack
    - function prologue:

```
pushl %ebp
movl %esp, %ebp
```

      or

```
enter $0, $0
```

      enter usually not used: 4 bytes vs 3 for pushl+movl, not on hardware fast-path anymore
    - function epilogue can easily find return EIP on stack:

```
movl %ebp, %esp
popl %ebp
```

      or

```
leave
```

      leave used often because it's 1 byte, vs 3 for movl+popl
- Big example:
    - C code

```
int main(void) { return f(8)+1; }
int f(int x) { return g(x); }
int g(int x) { return x+3; }
```

    - assembler

```
_main:
                            prologue
        pushl %ebp
```

```
        movl %esp, %ebp
                        body
        pushl $8
        call _f
        addl $1, %eax
                        epilogue
        movl %ebp, %esp
        popl %ebp
        ret
_f:
                        prologue
        pushl %ebp
        movl %esp, %ebp
                        body
        pushl 8(%esp)
        call _g
                        epilogue
        movl %ebp, %esp
        popl %ebp
        ret

_g:
                        prologue
        pushl %ebp
        movl %esp, %ebp
                        save %ebx
        pushl %ebx
                        body
        movl 8(%ebp), %ebx
        addl $3, %ebx
        movl %ebx, %eax
                        restore %ebx
        popl %ebx
                        epilogue
        movl %ebp, %esp
        popl %ebp
        ret
```

- Super-small `_g`:

```
_g:
        movl 4(%esp), %eax
        addl $3, %eax
        ret
```

- Shortest `_f`?
- Compiling, linking, loading:
  - *Preprocessor* takes C source code (ASCII text), expands #include etc, produces C source code
  - *Compiler* takes C source code (ASCII text), produces assembly language (also ASCII text)
  - *Assembler* takes assembly language (ASCII text), produces `.o` file (binary, machine-readable!)
  - *Linker* takes multiple '`.o`'s, produces a single *program image* (binary)
  - *Loader* loads the program image into memory at run-time and starts it executing

PC emulation

- The Bochs emulator works by
  - doing exactly what a real PC would do,

- only implemented in software rather than hardware!
- Runs as a normal process in a "host" operating system (e.g., Linux)
- Uses normal process storage to hold emulated hardware state: e.g.,
    - Stores emulated CPU registers in global variables

```
int32_t regs[8];
#define REG_EAX 1;
#define REG_EBX 2;
#define REG_ECX 3;
...
int32_t eip;
int16_t segregs[4];
...
```

    - Stores emulated physical memory in Boch's memory

```
char mem[256*1024*1024];
```

- Execute instructions by simulating them in a loop:

```
for (;;) {
        read_instruction();
        switch (decode_instruction_opcode()) {
        case OPCODE_ADD:
                int src = decode_src_reg();
                int dst = decode_dst_reg();
                regs[dst] = regs[dst] + regs[src];
                break;
        case OPCODE_SUB:
                int src = decode_src_reg();
                int dst = decode_dst_reg();
                regs[dst] = regs[dst] - regs[src];
                break;
        ...
        }
        eip += instruction_length;
}
```

- Simulate PC's physical memory map by decoding emulated "physical" addresses just like a PC would:

```
#define KB              1024
#define MB              1024*1024

#define LOW_MEMORY      640*KB
#define EXT_MEMORY      10*MB

uint8_t low_mem[LOW_MEMORY];
uint8_t ext_mem[EXT_MEMORY];
uint8_t bios_rom[64*KB];

uint8_t read_byte(uint32_t phys_addr) {
        if (phys_addr < LOW_MEMORY)
                return low_mem[phys_addr];
        else if (phys_addr >= 960*KB && phys_addr < 1*MB)
                return rom_bios[phys_addr - 960*KB];
        else if (phys_addr >= 1*MB && phys_addr < 1*MB+EXT_MEMORY) {
                return ext_mem[phys_addr-1*MB];
        else ...
}

void write_byte(uint32_t phys_addr, uint8_t val) {
        if (phys_addr < LOW_MEMORY)
                low_mem[phys_addr] = val;
```

```
        else if (phys_addr >= 960*KB && phys_addr < 1*MB)
                ; /* ignore attempted write to ROM! */
        else if (phys_addr >= 1*MB && phys_addr < 1*MB+EXT_MEMORY) {
                ext_mem[phys_addr-1*MB] = val;
        else ...
}
```

- Simulate I/O devices, etc., by detecting accesses to "special" memory and I/O space and emulating the correct behavior: e.g.,
  - Reads/writes to emulated hard disk transformed into reads/writes of a file on the host system
  - Writes to emulated VGA display hardware transformed into drawing into an X window
  - Reads from emulated PC keyboard transformed into reads from X input event queue

```
6.828 2017 Lecture 4: Shell & OS organization

Lecture Topic:
  kernel system call API
    both details and design
  illustrate via shell and homework 2

Overview Diagram
  user / kernel
  process = address space + thread(s)
  app -> printf() -> write() -> SYSTEM CALL -> sys_write() -> ...
  user-level libraries are app's private business
  kernel internal functions are not callable by user
  xv6 has a few dozen system calls; Linux a few hundred
  details today are mostly about UNIX system-call API
    basis for xv6, Linux, OSX, POSIX standard, &c
    jos has very different system-calls; you'll build UNIX calls over jos

Homework solution

* Let's review Homework 2 (sh.c)
  * exec
    why two execv() arguments?
    what happens to the arguments?
    what happens when exec'd process finishes?
    can execv() return?
    how is the shell able to continue after the command finishes?
  * redirect
    how does exec'd process learn about redirects? [kernel fd tables]
    does the redirect (or error exit) affect the main shell?
  * pipe
    ls | wc -l
    what if ls produces output faster than wc consumes it?
    what if ls is slower than wc?
    how does each command decide when to exit?
    what if reader didn't close the write end? [try it]
    what if writer didn't close the read end?
    how does the kernel know when to free the pipe buffer?

  * how does the shell know a pipeline is finished?
    e.g. ls | sort | tail -1

  * what's the tree of processes?
    sh parses as: ls | (sort | tail -1)
          sh
          sh1
      ls      sh2
          sort   tail

  * does the shell need to fork so many times?
    - what if sh didn't fork for pcmd->left? [try it]
      i.e. called runcmd() without forking?
    - what if sh didn't fork for pcmd->right? [try it]
      would user-visible behavior change?
      sleep 10 | echo hi

  * why wait() for pipe processes only after both are started?
    what if sh wait()ed for pcmd->left before 2nd fork? [try it]
      ls | wc -l
      cat < big | wc -l

  * the point: the system calls can be combined in many ways
    to obtain different behaviors.

Let's look at the challenge problems

 * How to implement sequencing with ";"?
   gcc sh.c ; ./a.out
   echo a ; echo b
```

   why wait() before scmd->right? [try it]

 * How to implement "&"?
   $ sleep 5 &
   $ wait
   the implementation of & and wait is in main -- why?
   What if a background process exits while sh waits for a foreground process?

 * How to implement nesting?
   $ (echo a; echo b) | wc -l
   my (...) implementation is only in sh's parser, not runcmd()
   it's neat that sh pipe code doesn't have to know it's applying to a sequence

 * How do these differ?
   echo a > x ; echo b > x
   (echo a ; echo b) > x
   what's the mechanism that avoids overwriting?

UNIX system call observations

* The fork/exec split looks wasteful -- fork() copies mem, exec() discards.
  why not e.g. pid = forkexec(path, argv, fd0, fd1) ?
  the fork/exec split is useful:
    fork(); I/O redirection; exec()
      or fork(); complex nested command; exit.
      as in (cmd1 ; cmd2) | cmd3
    fork() alone: parallel processing
    exec() alone: /bin/login ... exec("/bin/sh")
  fork is cheap for small programs -- on my machine:
    fork+exec takes 400 microseconds (2500 / second)
    fork alone takes 80 microseconds (12000 / second)
    some tricks are involved -- you'll implement them in jos!

* The file descriptor design:
  * FDs are a level of indirection
    - a process's real I/O environment is hidden in the kernel
    - preserved over fork and exec
    - separates I/O setup from use
    - imagine writefile(filename, offset, buf size)
  * FDs help make programs more general purpose: don't need special cases for
    files vs console vs pipe

* Philosophy: small set of conceptually simple calls that combine well
  e.g. fork(), open(), dup(), exec()
  command-line design has a similar approach
    ls | wc -l

* Why must kernel support pipes -- why not have sh simulate them, e.g.
  ls > tempfile ; wc -l < tempfile

* System call interface simple, just ints and char buffers. why not have open()
  return a pointer reference to a kernel file object?

* The core UNIX system calls are ancient; have they held up well?
  yes; very successful
    and evolved well over many years
  history: design caters to command-line and s/w development
    system call interface is easy for programmers to use
    command-line users like named files, pipelines, &c
    important for development, debugging, server maintenance
  but the UNIX ideas are not perfect:
    programmer convenience is often not very valuable for system-call API
      programmers use libraries e.g. Python that hide sys call details
      apps may have little to do with files &c, e.g. on smartphone
    some UNIX abstractions aren't very efficient
      fork() for multi-GB process is very slow
      FDs hide specifics that may be important
        e.g. block size for on-disk files
        e.g. timing and size of network messages

```
  so there has been lots of work on alternate plans
    sometimes new system calls and abstractions for existing UNIX-like kernels
    sometimes entirely new approaches to what a kernel should do
  ask "why this way? wouldn't design X be better?"

OS organization

* How to implement a system-call interface?

* Why not just a library?
  I.e. no kernel, just run app+library directly on the hardware.
  flexible: apps can bypass library if it's not right
  apps can directly interact with hardware
  a library is OK for a single-purpose device
  but what if the computer is used for multiple activities?

* Key requirements for kernels:
  isolation
  multiplexing
  interaction

* helpful approach: abstract resources rather than raw hardware
  File system, not raw disk
  Processes, not raw CPU/memory
  TCP, not ethernet packets
  abstractions often ease isolation, multiplexing and interaction
  also more convenient and portable

* Start with isolation since that's often the most constraining requirement.

* Isolation goals:
  apps cannot directly interact with hardware
  apps cannot harm operating system
  apps cannot directly affect each other
  apps can only interact with world via the OS interface

* Processors provide mechanisms that help with isolation
  * Hardware provides user mode and kernel mode
    - some instructions can only be executed in kernel mode
      device access, processor configuration, isolation mechanisms
  * Hardware forbids apps from executing privileged instructions
    - instead traps to kernel mode
    - kernel can clean up (e.g., kill the process)
  * Hardware lets kernel mode configure various constraints on user mode
    most critical: page tables to limit user s/w to its own address space

* Kernel builds on hardware isolation mechanisms

  * Operating system runs in kernel mode
    - kernel is a big program
      services: processes, file system, net
      low-level: devices, virtual memory
      all of kernel runs with full hardware privilege (convenient)

  * Applications run in user mode
    - kernel sets up per-process isolated address space
    - system calls switch between user and kernel mode
      the application executes a special instruction to enter kernel
      hardware switches to kernel mode
      but only at an entry point specified by the kernel

* What to put in the kernel?

  * xv6 follows a traditional design: all of the OS runs in kernel mode
    - one big program with file system, drivers, &c
    - this design is called a monolithic kernel
    - kernel interface == system call interface
    - good: easy for subsystems to cooperate
      one cache shared by file system and virtual memory
```

```
    - bad: interactions are complex
      leads to bugs
      no isolation within kernel

  * microkernel design
    - many OS services run as ordinary user programs
      file system in a file server
    - kernel implements minimal mechanism to run services in user space
      processes with memory
      inter-process communication (IPC)
    - kernel interface != system call interface
    - good: more isolation
    - bad: may be hard to get good performance

  * exokernel: no abstractions
    apps can use hardware semi-directly, but O/S isolates
    e.g. app can read/write own page table, but O/S audits
    e.g. app can read/write disk blocks, but O/S tracks block owners
    good: more flexibility for demanding applications
    jos will be a mix of microkernel and exokernel

* Can one have process isolation WITHOUT h/w-supported kernel/user mode?
  yes!
  see Singularity O/S, later in semester
  but h/w user/kernel mode is the most popular plan

Next lecture: x86 hardware isolation mechanisms and xv6's use of them
```

```
6.828 2017 Lecture 5: Isolation mechanisms

Today:
  user/kernel isolation
  xv6 system call as case study

* How to choose overall form for a kernel?
  many possible answers!
  one extreme:
    just a library of device drivers, linked w/ app
    run application directly on hardware
    fast and flexible for single-purpose devices
    but usually multiple tasks on a computer

* Multiple tasks drive the key requirements:
  multiplexing
  isolation
  interaction

* helpful approach: abstract resources rather than raw hardware
  File system, not raw disk
  Processes, not raw CPU/memory
  TCP connections, not ethernet packets
  abstractions are often easier to isolate and share
    e.g. programs see a private CPU, needn't think about multiplexing
  also more convenient and portable

* Isolation is often the most constraining requirement.

* What is isolation?
  enforced separation to contain effects of failures
  the process is the usual unit of isolation
  prevent process X from wrecking or spying on process Y
    r/w memory, use 100% of CPU, change FDs, &c
  prevent a process from interfering with the operating system
  in the face of malice as well as bugs
    a bad process may try to trick the h/w or kernel

* the kernel uses hardware mechanisms as part of process isolation:
  user/kernel mode flag
  address spaces
  timeslicing
  system call interface

* the hardware user/kernel mode flag
  controls whether instructions can access privileged h/w
  called CPL on the x86, bottom two bits of %cs register
    CPL=0 -- kernel mode -- privileged
    CPL=3 -- user mode -- no privilege
  x86 CPL protects many processor registers relevant to isolation
    I/O port accesses
    control register accesses (eflags, %cs4, ...)
      including %cs itself
    affects memory access permissions, but indirectly
    the kernel must set all this up correctly
  every serious microprocessor has some kind of user/kernel flag

* 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 ("vectors")
    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)

* 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 code
    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" in the memory management unit (MMU)
  MMU translates (or "maps") every address issued by program
```
    CPU -> MMU -> RAM
            |
         pagetable
    VA -> PA
```
    MMU translates all memory references: user and kernel, instructions and data
    instructions use only VAs, never PAs
  kernel sets up a different page table for each process
    each process's page table allows access only to that process's RAM

Let's look at how xv6 system calls are implemented

xv6 process/stack diagram:
  user process ; kernel thread
  user stack ; kernel stack
  two mechanisms:
    switch between user/kernel
    switch between kernel threads
  trap frame
  kernel function calls...
  struct context

* simplified xv6 user/kernel virtual address-space setup
```
  FFFFFFFF:
            ...
  80000000: kernel
            user stack
            user data
  00000000: user instructions
```
  kernel configures MMU to give user code access only to lower half
  separate address space for each process
    but kernel (high) mappings are the same for every process

system call starting point:
  executing in user space, sh writing its prompt
  sh.asm, write() library function
  break *0xb90
  x/3i 0xb8b
    0x10 in eax is the system call number for write
  info reg
    cs=0x1b, B=1011 -- CPL=3 => user mode
    esp and eip are low addresses -- user virtual addresses
  x/4x $esp
    cc1 is return address -- in printf
    2 is fd
    0x3f7a is buffer on the stack
    1 is count
    i.e. write(2, 0x3f7a, 1)
  x/c 0x3f7a

INT instruction, kernel entry

```
  stepi
  info reg
    cs=0x8 -- CPL=3 => kernel mode
    note INT changed eip and esp to high kernel addresses
  where is eip?
    at a kernel-supplied vector -- only place user can go
    so user program can't jump to random places in kernel with CPL=0
  x/6wx $esp
    INT saved a few user registers
    err, eip, cs, eflags, esp, ss
  why did INT save just these registers?
    they are the ones that INT overwrites
  what INT did:
    switched to current process's kernel stack
    saved some user registers on kernel stack
    set CPL=0
    start executing at kernel-supplied "vector"
  where did esp come from?
    kernel told h/w what kernel stack to use when creating process

Q: why does INT bother saving the user state?
   how much state should be saved?
   transparency vs speed

saving the rest of the user registers on the kernel stack
  trapasm.S alltraps
  pushal pushes 8 registers: eax .. edi
  x/19x $esp
  19 words at top of kernel stack:
    ss
    esp
    eflags
    cs
    eip
    err    -- INT saved from here up
    trapno
    ds
    es
    fs
    gs
    eax..edi
  will eventually be restored, when system call returns
  meanwhile the kernel C code sometimes needs to read/write saved values
  struct trapframe in x86.h

Q: why are user registers saved on the kernel stack?
   why not save them on the user stack?

entering kernel C code
  the pushl %esp creates an argument for trap(struct trapframe *tf)
  now we're in trap() in trap.c
  print tf
  print *tf

kernel system call handling
  device interrupts and faults also enter trap()
  trapno == T_SYSCALL
  myproc()
  struct proc in proc.h
  myproc()->tf -- so syscall() can get at call # and arguments
  syscall() in syscall.c
    looks at tf->eax to find out which system call
  SYS_write in syscalls[] maps to sys_write
  sys_write() in sysfile.c
  arg*() read write(fd,buf,n) arguments from the user stack
  argint() in syscall.c
    proc->tf->esp + xxx

restoring user registers
```

```
  syscall() sets tf->eax to return value
  back to trap()
  finish -- returns to trapasm.S
  info reg -- still in kernel, registers overwritten by kernel code
  stepi to iret
  info reg
    most registers hold restored user values
    eax has write() return value of 1
    esp, eip, cs still have kernel values
  x/5x $esp
    saved user state: eip, cs, eflags, esp, ss
  IRET pops those user registers from the stack
    and thereby re-enters user space with CPL=3

Q: do we really need IRET?
   could we use ordinary instructions to restore the registers?
   could IRET be simpler?

back to user space
  stepi
  info reg

*** fork()

let's look at how fork() sets up a new process
in particular, how to get the new process into user space the first time?
the idea:
  fork() fakes a kernel stack that *looks* like it's about to return from trap()
    with a faked trapframe at the top
  child starts executing in the kernel -- at a function return instruction
  alltraps "restores" faked saved registers
  starts executing the child for the first time

note there are two separate actions:
  create a new process
  execute the new process

break fork
c
where

fork() in proc.c

allocproc()
  look at proc[] at start of proc.c
  focus on initial content of p->kstack
  space for trap frame (will be a copy of parent's)
  fake saved EIP that points to trapret in trapasm.S
  kernel stack space for a "context"
    contains *kernel* registers
    to be restored when switching to child's kernel thread
    proc.h
  the p->context->eip = forkret sets up where child starts in the kernel
    basically just a fuction call instruction

back to fork()
  (remember we're still executing as the parent)
  allocate physical memory and a page table
  copy parent's memory to child
  copy trapframe
  tf->eax = 0 -- this will the child's return value from fork():w
  print *np
  print *np->tf
  print *np->context
  x/25x np->context
  state = RUNNABLE -- now we are done

the new process's kernel stack:
  trapframe -- copy of parent, but eax=0
```

```
  trapret's address
  context
    eip = forkret

break forkret
x/20x $esp
next
finish
(now in trapret in tramasm.S)
at b6a in sh.S
info reg
and eax is zero -- it's the child
```

6.828 2016 Lecture 6: Virtual Memory
==

* plan:
  address spaces
  paging hardware
  xv6 VM code
    case study
    finish lec 5
    homework sol

Virtual memory overview

* today's problem:
  [user/kernel diagram]
  [memory view: diagram with user processes and kernel in memory]
  suppose the shell has a bug:
    sometimes it writes to a random memory address
  how can we keep it from wrecking the kernel?
    and from wrecking other processes?

* we want isolated address spaces
  each process has its own memory
  it can read and write its own memory
  it cannot read or write anything else
  challenge:
    how to multiplex several memories over one physical memory?
        while maintaining isolation between memories

* xv6 and JOS uses x86's paging hardware to implement AS's
  ask questions! this material is important

* paging provides a level of indirection for addressing
  CPU -> MMU -> RAM
      VA   PA
  s/w can only ld/st to virtual addresses, not physical
  kernel tells MMU how to map each virtual address to a physical address
    MMU essentially has a table, indexed by va, yielding pa
    called a "page table"
  MMU can restrict what virtual addresses user code can use

* x86 maps 4-KB "pages"
  and aligned -- start on 4 KB boundaries
  thus page table index is top 20 bits of VA

* what is in a page table entry (PTE)?
  see [handout](x86_translation_and_registers.pdf)
  top 20 bits are top 20 bits of physical address
    "physical page number"
    MMU replaces top 20 of VA with PPN
  low 12 bits are flags
    Present, Writeable, &c

* where is the page table stored?
  in RAM -- MMU loads (and stores) PTEs
  o/s can read/write PTEs

* would it be reasonable for page table to just be an array of PTEs?
  how big is it?
  2^20 is a million
  32 bits per entry
  4 MB for a full page table -- pretty big on early machines
  would waste lots of memory for small programs!
    you only need mappings for a few hundred pages
    so the rest of the million entries would be there but not needed

* x86 uses a "two-level page table" to save space
  diagram
  pages of PTEs in RAM

```
  page directory (PD) in RAM
  PDE also contains 20-bit PPN -- of a page of 1024 PTEs
  1024 PDEs point to PTE pages
    each PTE page has 1024 PTEs -- so 1024*1024 PTEs in total
  PD entries can be invalid
    those PTE pages need not exist
    so a page table for a small address space can be small

* how does the mmu know where the page table is located in RAM?
  %cr3 holds phys address of PD
  PD holds phys address of PTE pages
  they can be anywhere in RAM -- need not be contiguous

* how does x86 paging hardware translate a va?
  need to find the right PTE
  %cr3 points to PA of PD
  top 10 bits index PD to get PA of PT
  next 10 bits index PT to get PTE
  PPN from PTE + low-12 from VA

* flags in PTE
  P, W, U
  xv6 uses U to forbid user from using kernel memory

* what if P bit not set? or store and W bit not set?
  "page fault"
  CPU saves registers, forces transfer to kernel
  trap.c in xv6 source
  kernel can just produce error, kill process
  or kernel can install a PTE, resume the process
    e.g. after loading the page of memory from disk

* Q: why mapping rather than e.g. base/bound?
  indirection allows paging h/w to solve many problems
  e.g. avoids fragmentation
  e.g. copy-on-write fork
  e.g. lazy allocation (home work for next lecture)
  many more techniques
  topic of next lecture

* Q: why use virtual memory in kernel?
  it is clearly good to have page tables for user processes
  but why have a page table for the kernel?
    could the kernel run with using only physical addresses?
  top-level answer: yes
    Singularity is an example kernel using phys addresses
        but, most standard kernels do use virtual addresses?
  why do standard kernels do so?
    some reasons are lame, some are better, none are fundamental
    - the hardware makes it difficult to turn it off
         e.g. on entering a system call, one would have to disable VM
        - it can be convenient for the kernel to use user addresses
         e.g. a user address passed to a system call
         but, probably a bad idea: poor isolation between kernel/application
        - convenient if addresses are contiguous
         say kernel has both 4Kbyte objects and 64Kbyte objects
      without page tables, we can easily have memory fragmentation
         e.g., allocate 64K, allocate 4Kbyte, free the 64K, allocate 4Kbyte from the 64Kbyte
         now a new 64Kbyte object cannot use the free 60Kbyte.
        - the kernel must run of a wide range of hardware
         they may have different physical memory layouts

## Case study: xv6 use of the x86 paging hardware

* big picture of an xv6 address space -- one per process
  [diagram]
  0x00000000:0x80000000 -- user addresses below KERNBASE
  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

* where does xv6 map these regions, in phys mem?
<!--
 diagram from book: xv6-layout.eps
-->
  note double-mapping of user pages

* each process has its own address space
  and its own page table
  all processes have the same kernel (high memory) mappings
  kernel switches page tables (i.e. sets %cr3) when switching processes

* Q: why this address space arrangement?
  user virtual addresses start at zero
    of course user va 0 maps to different pa for each process
  2GB for user heap to grow contiguously
    but needn't have contiguous phys mem -- no fragmentation problem
  both kernel and user mapped -- easy to switch for syscall, interrupt
  kernel mapped at same place for all processes
    eases switching between processes
  easy for kernel to r/w user memory
    using user addresses, e.g. sys call arguments
  easy for kernel to r/w physical memory
    pa x mapped at va x+0x80000000
    we'll see this soon while manipulating page tables

* Q: what's the largest process this scheme can accommodate?

* Q: could we increase that by increasing/decreasing 0x80000000?

* Q: does the kernel have to map all of phys mem into its virtual address space?

* let's look at some xv6 virtual memory code
  terminology: virtual memory == address space / translation
  will help you w. next homework and labs

<!---

start where Robert left off: first process

setup: CPUS=1, turn-off interrupts in lapic.c
b proc.c:297

p *p
Q: are these addresses virtual addresses

break into qemu: info pg (modified 6.828 qemu)

step into switchuvm

x/1024x p->pgdir
what is 0x0dfbc007? (pde; see handout)
what is 0x0dfbc000?
what is 0x0dfbc000 + 0x8000000
what is there? (pte)
what is at 0x8dfbd000?
x x/i 0x8dfbd000 (first word of initcode.asm)

step passed lcr3

qemu: info pg

-->

* where did this pgdir get setup?
  look at vm.c: setupkvm and inituvm

```
* mappages() in vm.c
  arguments are PD, va, size, pa, perm
  adds mappings from a range of va's to corresponding pa's
  rounds b/c some uses pass in non-page-aligned addresses
  for each page-aligned address in the range
    call walkpgdir to find address of PTE
      need the PTE's address (not just content) b/c we want to modify
    put the desired pa into the PTE
    mark PTE as valid w/ PTE_P

* diagram of PD &c, as following steps build it

* walkpgdir() in vm.c
  mimics how the paging h/w finds the PTE for an address
  refer to the handout
  PDX extracts top ten bits
  &pgdir[PDX(va)] is the address of the relevant PDE
  now *pde is the PDE
  if PTE_P
    the relevant page-table page already exists
    PTE_ADDR extracts the PPN from the PDE
    p2v() adds 0x80000000, since PTE holds physical address
  if not PTE_P
    alloc a page-table page
    fill in PDE with PPN -- thus v2p
  now the PTE we want is in the page-table page
    at offset PTX(va)
    which is 2nd 10 bits of va


<!--

finish starting the first user process

return to gdb

(draw picture of kstack)
p /x p->tf
p /x *p->tf
p /x p->context
p /x p->context

b *0x0

swtch
x/8x $esp
forkret
x/19x $esp
info reg

step till user space:
x/i 0x0

step through use code
trap into kernel

x/19x $esp

-->

* tracing and date system call

<!-- homework
syscall trace
  syscall.c (HWSYS)
  return value in eax
  use STAB for printing out names
date
  usys.S
```

  syscall.c (HWDATE)
  argptr
-->

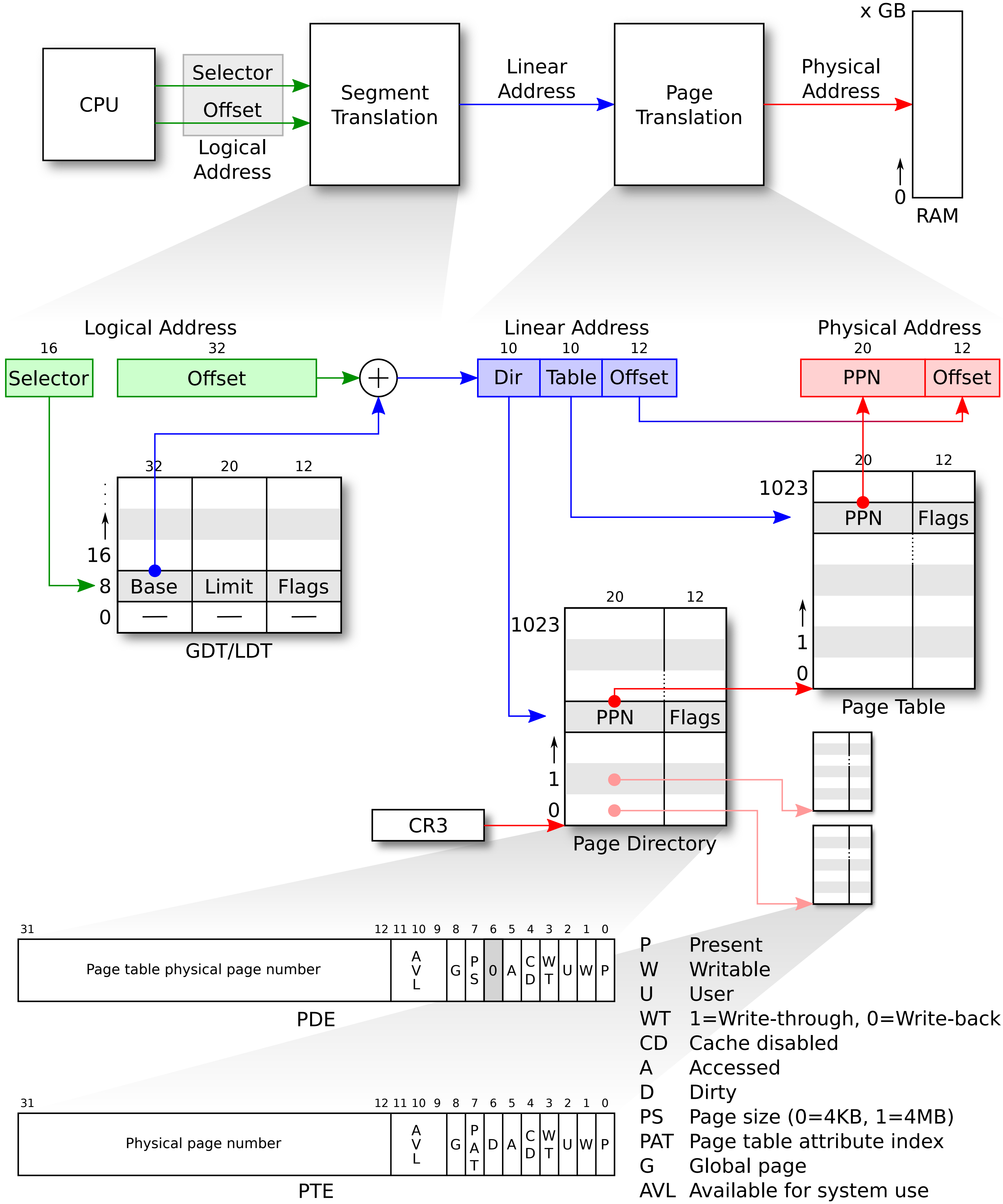
* a process calls sbrk(n) to ask for n more bytes of heap memory
  malloc() uses sbrk()
  each process has a size
    kernel adds new memory at process's end, increases size
  sbrk() allocates physical memory (RAM)
  maps it into the process's page table
  returns the starting address of the new memory

* sys_sbrk() in sysproc.c
<!---
   trace sbrk from user space
   just run ls (or any other cmd from shell)
   the new process forked by shell calls malloc for execcmd structure
   malloc.c calls sbrk
-->

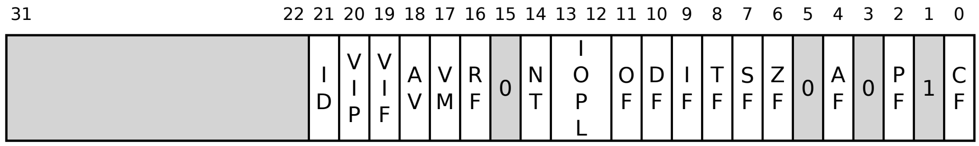
* growproc() in proc.c
  proc->sz is the process's current size
  allocuvm() does most of the work
  switchuvm sets %cr3 with new page table
    also flushes some MMU caches so it will see new PTEs

* allocuvm() in vm.c
  why if(newsz >= KERNBASE) ?
  why PGROUNDUP?
  arguments to mappages()...

Protected-Mode Address Translation



EFLAGS

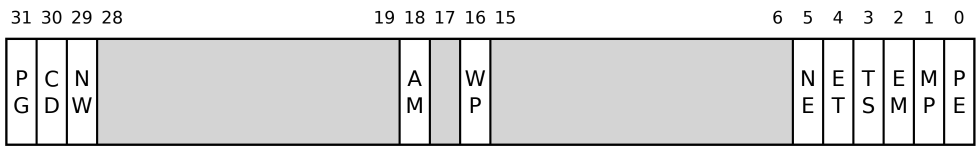


Status flags
CF Carry flag
PF Parity flag
AF Auxiliary carry flag
ZF Zero flag
SF Sign flag
OF Overflow flag

Control flags
DF Direction flag
System flags
TF Trap flag
IF Interrupt enable flag
IOPL I/O privilege level
NT Nested task

RF Resume flag
VM Virtual-8086 mode
AC Alignment check
VIF Virtual intr. flag
VIP Virtual intr. pending
ID ID flag

CR0

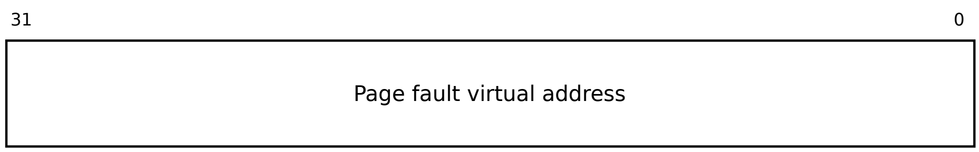


PE Protection enabled
MP Monitor coprocessor
EM Emulation
TS Task switched

ET Extension type
NE Numeric error
WP Write protect
AM Alignment mask

NW Not write-through
CD Cache disable
PG Paging

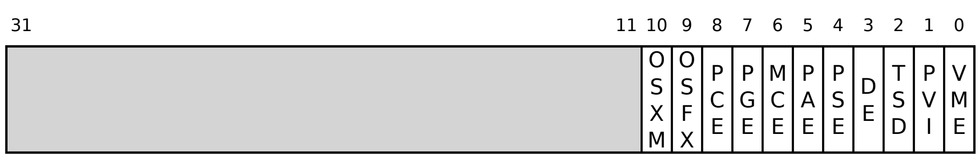
CR2



CR3

PWT Page-level writes transparent
PCT Page-level cache disable

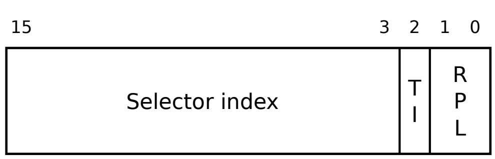
CR4



VME Virtual-8086 mode extensions
PVI Protected-mode virtual interrupts
TSD Time stamp disable
DE Debugging extensions
PSE Page size extensions
PAE Physical-address extension

MCE Machine check enable
PGE Page-global enable
PCE Performance counter enable
OSFXSR OS FXSAVE/FXRSTOR support
OSXMM-EXCPT OS unmasked exception support

CS/DS/ES
FS/GS/SS



TI Table index (0=GDT, 1=LDT)
RPL Requester privilege level

6.828 2016 Lecture 7: using virtual memory
==

* plan: cool things you can do with vm
  - Better performance/efficiency
    e.g., one zero-filled page
        e.g., copy-on-write fork
  - New features
    e.g., memory-mapped files
  - JOS and VM
  - This lecture may generate ideas for last lab (final project)

<!--
  isolation: picture with walls

  return user space, until we hit first system call
    then switch to date homework

  date system call homework
    point out some of the walls:
          U/K bit
        user cannot execute privileged instructions
                user enter kernel only through system calls
            only kernel can load cr3
          Page tables
            no U bit on kernel pages
          But sharing too:
            Kernel can read/write user memory
            Requires kernel checks arguments of system call
-->

* virtual memory: several views
  * primary purpose: isolation
    each process has its own address space
  * Virtual memory provides a level-of-indirection
    provides kernel with opportunity to do cool stuff

* lazy/on-demand page allocation
  * sbrk() is old fashioned;
    it asks application to "predict" how much memory they need
        difficult for applications to predict how much memory they need in advance
        sbrk allocates memory that may never be used.
  * moderns OSes allocate memory lazily
    allocate physical memory when application needs it
  * HW solution
    <!---
          draw xv6 user-part of address space
          demo solution; breakpoint right before mappages in trap.c
      explain page faults
        -->

<!--
        xv6 memlayout discussion

  user virtual addresses start at zero
    of course user va 0 maps to different pa for each process
  2GB for user heap to grow contiguously
    but needn't have contiguous phys mem -- no fragmentation problem
  both kernel and user mapped -- easy to switch for syscall, interrupt
  kernel mapped at same place for all processes
    eases switching between processes
  easy for kernel to r/w user memory
    using user addresses, e.g. sys call arguments
  easy for kernel to r/w physical memory
    pa x mapped at va x+0x80000000
    we'll see this soon while manipulating page tables

  lame part: user stack
    also, initcode and date (different AS layout)

```
      but convenient to check if an address is valid (va < p->size)

  why is kernel using vm?
-->

* Step back: class perspective
  - There is no one best way to design an OS
    Many OSes use VM, but you don't have to
  - Xv6 and JOS present examples of OS designs
    They lack many features of sophisticated designs
    In fact, they are pretty lame compared to a real OS
      Yet, still quite complex
  - Our goal: to teach you the key ideas so that you can extrapolate
    Xv6 and JOS are minimal design to expose key ideas
      You should be able to make them better
      You should be able to dive into Linux and find your way

* guard page to protect against stack overflow
  * put a non-mapped page below user stack
    if stack overflows, application will see page fault
  * allocate more stack when application runs off stack into guard page
    <!---
       draw xv6 user-part of address space
     compile with -O so the compiler doesn't optimize the tail recursion
       demo stackoverflow
      set breakpoint at g
        run stackoverflow
        look at $esp
        look at pg info at qemu console
        note page has no U bit
      -->

* one zero-filled page
  * kernel often fills a page with zeros
  * idea: memset *one* page with zeros
    map that page copy-on-write when kernel needs zero-filled page
    on write make copy of page and map it read/write in app address space

* share kernel page tables in xv6
  * observation:
    kvmalloc() allocates new pages for kernel page table for each process
    but all processes have the same kernel page table
  * idea: modify kvmalloc()/freevm() to share kernel page table
    <!---
       demo HWKVM
      -->

* copy-on-write fork
  * observation:
    xv6 fork copies all pages from parent (see fork())
    but fork is often immediately followed by exec
  * idea: share address space between parent and child
    modify fork() to map pages copy-on-write (use extra available system bits in PTEs and PDEs)
    on page fault, make copy of page and map it read/write

* demand paging
  * observation: exec loads the complete file into memory (see exec.c)
    expensive: takes time to do so (e.g., file is stored on a slow disk)
    unnecessary: maybe not the whole file will be used
  * idea: load pages from the file on demand
    allocate page table entries, but mark them on-demand
    on fault, read the page in from the file and update page table entry
  * challenge: file larger than physical memory (see next idea)

* use virtual memory larger than physical memory
  * observation: application may need more memory than there is physical memory
  * idea: store less-frequently used parts of the address space on disk
    page-in and page-out pages of the address address space transparently
  * works when working sets fits in physical memory
```

```
* memory-mapped files
  * idea: allow access to files using load and store
    can easily read and writes part of a file
    e.g., don't have to change offset using lseek system call
  * page-in pages of a file on demand
    when memory is full, page-out pages of a file that are not frequently used

* shared virtual memory
  * idea: allow processes on different machines to share virtual memory
    gives the illusion of physical shared memory, across a network
  * replicate pages that are only read
  * invalidate copies on write

* JOS and virtual memory
  * layout: [picture](l-josmem.html)

  * UVPT trick (lab 4)
    recursively map PD at 0x3BD
      virtual address of PD is (0x3BD<<22)|(0x3BD<<12)
    if we want to find pte for virtual page n, compute
                pde_t uvpt[n], where uvpt is (0x3BD << 22)
        = uvpt + n * 4 (because pdt is a word)
            = (0x3BD << 22) | (top 10 bits of n) | (bottom 10 bits of n) << 2
                = 10 | 10 | 12
    for example, uvpt[0] is address (0x3BD << 22), following the pointers gives us
        the first entry in the page directory, which points to the first page table, which
        we index with 0, which gives us pte 0

    simpler than pgdirwalk()?

  * user-level copy-on-write fork (lab4)
    JOS propagates page faults to user space
    user programs can play similar VM tricks as kernel!
    you will do user-level copy-on-write fork
```

How we will use paging (and segments) in JOS:

- use segments only to switch privilege level into/out of kernel
- use paging to structure process address space
- use paging to limit process memory access to its own address space
- below is the JOS virtual memory map
- why map both kernel and current process? why not 4GB for each? how does this compare with xv6?
- why is the kernel at the top?
- why map all of phys mem at the top? i.e. why multiple mappings?
- (will discuss UVPT in a moment...)
- how do we switch mappings for a different process?

```
    4 Gig -------->  +------------------------------+
                     |                              | RW/--
                     ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                     :              .               :
                     :              .               :
                     :              .               :
                     |~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~| RW/--
                     |                              | RW/--
                     |   Remapped Physical Memory   | RW/--
                     |                              | RW/--
    KERNBASE, ---->  +------------------------------+ 0xf0000000      --+
    KSTACKTOP        |     CPU0's Kernel Stack      | RW/--  KSTKSIZE   |
                     | - - - - - - - - - - - - - - -|                   |
                     |      Invalid Memory (*)      | --/--  KSTKGAP    |
                     +------------------------------+                   |
                     |     CPU1's Kernel Stack      | RW/--  KSTKSIZE   |
                     | - - - - - - - - - - - - - - -|                 PTSIZE
                     |      Invalid Memory (*)      | --/--  KSTKGAP    |
                     +------------------------------+                   |
                     :              .               :                   |
                     :              .               :                   |
    MMIOLIM ------>  +------------------------------+ 0xefc00000      --+
                     |       Memory-mapped I/O      | RW/--  PTSIZE
 ULIM, MMIOBASE -->  +------------------------------+ 0xef800000
                     |  Cur. Page Table (User R-)   | R-/R-  PTSIZE
    UVPT      ---->  +------------------------------+ 0xef400000
                     |          RO PAGES            | R-/R-  PTSIZE
    UPAGES    ---->  +------------------------------+ 0xef000000
                     |           RO ENVS            | R-/R-  PTSIZE
 UTOP,UENVS ------>  +------------------------------+ 0xeec00000
 UXSTACKTOP -/       |     User Exception Stack     | RW/RW  PGSIZE
                     +------------------------------+ 0xeebff000
                     |       Empty Memory (*)       | --/--  PGSIZE
    USTACKTOP  --->  +------------------------------+ 0xeebfe000
                     |      Normal User Stack       | RW/RW  PGSIZE
                     +------------------------------+ 0xeebfd000
                     |                              |
                     |                              |
                     ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                     .                              .
                     .                              .
                     .                              .
                     |~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~|
                     |     Program Data & Heap      |
    UTEXT -------->  +------------------------------+ 0x00800000
    PFTEMP ------->  |       Empty Memory (*)       |        PTSIZE
                     |                              |
    UTEMP -------->  +------------------------------+ 0x00400000      --+
                     |       Empty Memory (*)       |                   |
                     | - - - - - - - - - - - - - - -|                   |
```

```
                   |  User STAB Data (optional)   |                PTSIZE
USTABDATA ---->  +------------------------------+ 0x00200000       |
                   |       Empty Memory (*)       |                  |
0 ------------>  +------------------------------+                --+
```

The UVPT

We had a nice conceptual model of the page table as a 2^20-entry array that we could index with a physical page number. The x86 2-level paging scheme broke that, by fragmenting the giant page table into many page tables and one page directory. We'd like to get the giant conceptual page-table back in some way -- processes in JOS are going to look at it to figure out what's going on in their address space. But how?

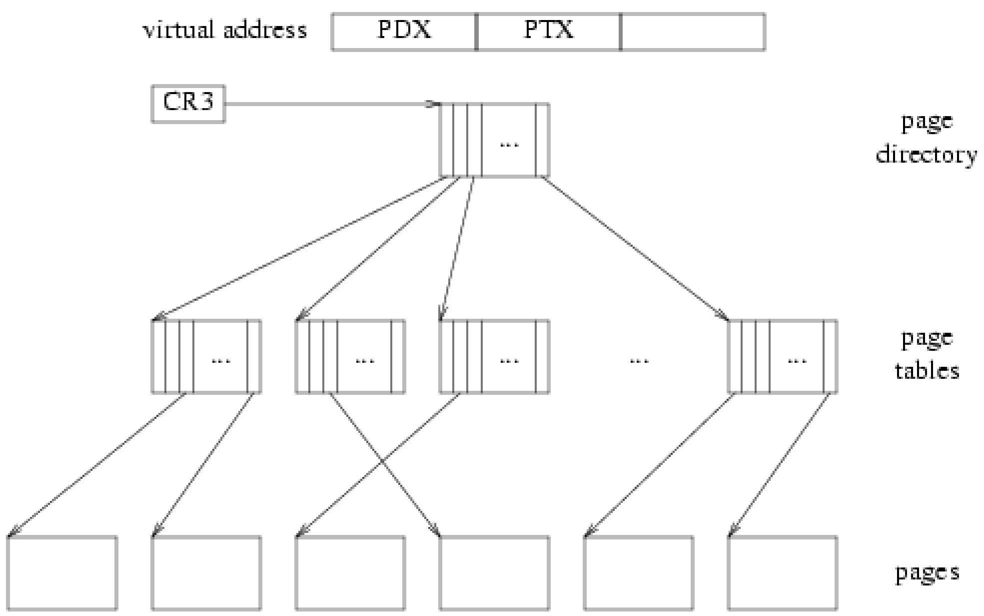
Luckily, the paging hardware is great for precisely this -- putting together a set of fragmented pages into a contiguous address space. And it turns out we already have a table with pointers to all of our fragmented page tables: it's the page directory!

So, we can use the page *directory* as a page *table* to map our conceptual giant 2^22-byte page table (represented by 1024 pages) at some contiguous 2^22-byte range in the virtual address space. And we can ensure user processes can't modify their page tables by marking the PDE entry as read-only.

Puzzle: do we need to create a separate UVPD mapping too?

A more detailed way of understanding this configuration:

Remember how the X86 translates virtual addresses into physical ones:

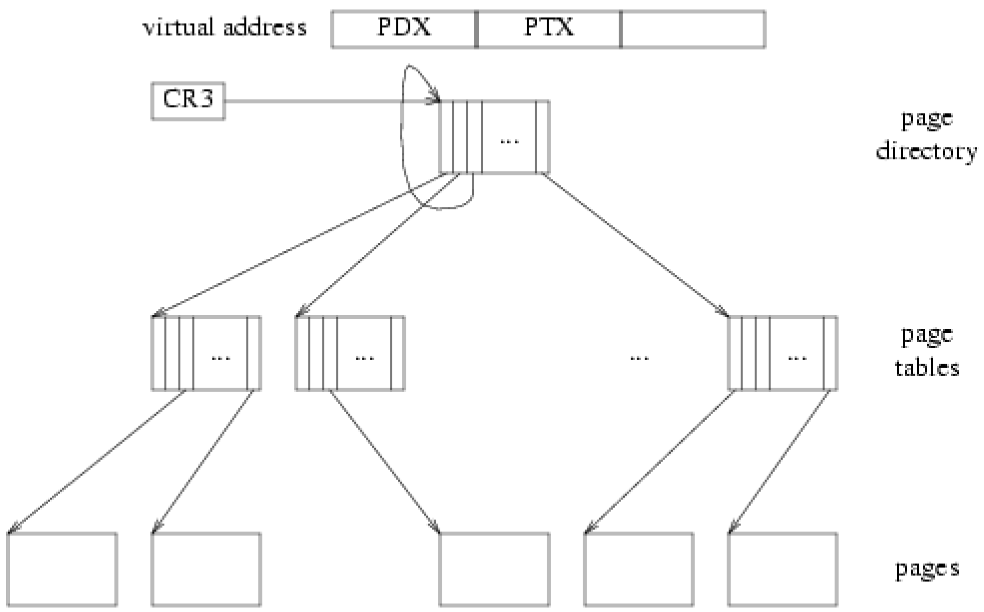


CR3 points at the page directory. The PDX part of the address indexes into the page directory to give you a page table. The PTX part indexes into the page table to give you a page, and then you add the low bits in.

But the processor has no concept of page directories, page tables, and pages being anything other than plain memory. So there's nothing that says a particular page in memory can't serve as two or three of these at once. The processor just follows pointers: pd = lcr3(); pt = *(pd+4*PDX); page = *(pt+4*PTX);

Diagramatically, it starts at CR3, follows three arrows, and then stops.

If we put a pointer into the page directory that points back to itself at index V, as in



then when we try to translate a virtual address with PDX and PTX equal to V, following three arrows leaves us at the page directory. So that virtual page translates to the page holding the page directory. In Jos, V is 0x3BD, so the virtual address of the UVPD is (0x3BD<<22)|(0x3BD<<12).

Now, if we try to translate a virtual address with PDX = V but an arbitrary PTX != V, then following three arrows from CR3 ends one level up from usual (instead of two as in the last case), which is to say in the page tables. So the set of virtual pages with PDX=V form a 4MB region whose page contents, as far as the processor is concerned, are the page tables themselves. In Jos, V is 0x3BD so the virtual address of the UVPT is (0x3BD<<22).

So because of the "no-op" arrow we've cleverly inserted into the page directory, we've mapped the pages being used as the page directory and page table (which are normally virtually invisible) into the virtual address space.

6.828 2017 Lecture 8: System calls, Interrupts, and Exceptions

Let's start with the homework

```
alarmtest.c
  alarm(10, periodic)
  asks kernel to call periodic() every 10 "ticks" in this process
  that is, every 10 ticks of CPU time that this process consumes
  three pieces:
    add a new system call
    count ticks as the program runs (timer interrupt)
    kernel "upcall" to periodic()
  the call to periodic() is a simplified UNIX signal

glue for a new system call
  syscall.h: #define SYS_alarm 22
  usys.S: SYSCALL(alarm)
    alarmtest.asm -- mov $0x16,%eax -- 0x16 is SYS_alarm
  syscall.c syscalls[] table
  sysproc.c sys_alarm()

why all this machinery?
  at a high level, alarmtest just wants to make a function call to sys_alarm
  it has to be indirect (via INT, SYS_alarm) to maintain isolation

break sys_alarm
  where
  how did syscall know which system call?
    trapframe, on kernel stack, has saved user eax
    print myproc()->tf->eax
  where does sys_alarm find the arguments, ticks and handler?
    on the user stack
    x/4x myproc()->tf->esp
  does the handler value make sense? look in alarmtest.asm

now we need to take some action whenever the timer h/w interrupts
  decrement ticksleft
  if expired
    upcall to handler (periodic())
    reset ticksleft

device interrupts arrive just like INT and pagefault
  h/w pushes esp and eip on kernel stack
  s/w saves other registers, into a trapframe
  vector, alltraps, trap()

timer interrupts served by IRQ_TIMER case in trap()
  original IRQ_TIMER task is to keep track of wall-clock time, in ticks

execute to trap without an implementation
  break vector32
  where
  print/x tf->eip
  print/x tf->esp
  x/4x tf->esp

what was the user program doing at this point?
  tf->eip in alarmtest.asm
  user code could have been interrupted anywhere
    so we can't rely on anything about the user stack
    and we need to restore registers exactly, since program didn't save anything

Q: how to arrange for upcall to alarm handler?
   call myproc()->alarmhandler() ?
   tf->eip = myproc()->alarmhandler ?
Q: how to ensure handler returns to interrupted user code?

add our code...
run alarmtest without gdb
```

```
let's run with gdb
  list trap to find breakpoint
  print/x tf->eip before assignment
  print/x tf->eip after assignment
  break *0x74
  c
  info reg
  will it return somewhere reasonable in alarmtest.asm?
  x/4x $esp

Q: what's the security problem in my new trap() code?

Q: what if trap() directly called alarmhandler()?
   it's a bad idea
   but what exactly would go wrong?
   let's try it
     it doesn't crash!
     but it doesn't print alarm! either. why not?
     fetchint...
   apparently it gets back to user space (to print .) -- how?
     program, timer trap, alarmhandler(), INT, sys_write("alarm!"), return...
     stack diagram

it is disturbing how close this came to working!
  why can kernel code directly jump to user instructions?
  why can user instructions modify the kernel stack?
  why do system calls (INT) work from the kernel?
  none of these are intended properties of xv6!
  the x86 h/w does *not* directly provide isolation
    x86 has many separate features (page table, INT, &c)
    it's possible to configure these features to enforce isolation
    but isolation is not the default!

Q: what happens if just tf->eip = alarmhandler, but don't push old eip?
   let's try it
   user stack diagram

Q: what if trap() didn't check for CPL 3?
   let's try it -- seems to work!
   how could tf->cs&3 == 0 ever arise from alarmtest?
   let's force the situation with (tf->cs&3)==0
     and making alarmtest run forever
     unexpected trap 14 from cpu 0 eip 801067cb (cr2=0x801050cf)
   what is eip 0x801067cb in kernel.asm?
     tf->esp = tf->eip in trap().
   what happened?
     it was a CPL=0 to CPL=0 interrupt
     so the h/w didn't switch stacks
     so it didn't save %esp
     so tf->esp contains garbage
     (see comment at end of trapframe in x86.h)
   the larger point is that interrupts can occur while in the kernel (in xv6, not JOS)

Q: what will happen if user-supplied alarm handler fn points into the kernel?
   (with the correct trap() code)

Q: what if another timer interrupt goes off while in user handler?
   works, but confusing, and will eventually run out of user stack
   maybe kernel shouldn't re-start timer until handler function finishes

Q: is it a problem if periodic() modifies registers?
   how could we arrange to restore registers before returning?

let's step back and talk about interrupts a bit more generally

the general topic: h/w wants attention now!
  s/w must set aside current work and respond
```

```
where do traps come from?
  (I use "trap" as a general term)
  device -- data ready, or completed an action, ready for more
  exception/fault -- page fault, divide by zero, &c
  INT -- system call
  IPI -- kernel CPU-to-CPU communication, e.g. to flush TLB

where do device interrupts come from?
  diagram:
    CPUs, LAPICs, IOAPIC, devices
    data bus
    interrupt bus
  the interrupt tells the kernel the device hardware wants attention
  the driver (in the kernel) knows how to tell the device to do things
  often the interrupt handler calls the relevant driver
    but other arrangements are possible (schedule a thread; poll)

how does trap() know which device interrupted?
  i.e. where did tf->trapno == T_IRQ0 + IRQ_TIMER come from?
  kernel tells LAPIC/IOAPIC what vector number to use, e.g. timer is vector 32
    page faults &c also have vectors
    LAPIC / IOAPIC are standard pieces of PC hardware
    one LAPIC per CPU
  IDT associates an instruction address with each vector number
    IDT format is defined by Intel, configured by kernel
  each vector jumps to alltraps
  CPU sends many kinds of traps through IDT
    low 32 IDT entries have special fixed meaning
  xv6 sets up system calls (IRQ) to use IDT entry 64 (0x40)
  the point: the vector number reveals the source of the interrupt

diagram:
  IRQ or trap, IDT table, vectors, alltraps
  IDT:
    0: divide by zero
    13: general protection
    14: page fault
    32-255: device IRQs
    32: timer
    33: keyboard
    46: IDE
    64: INT

let's look at how xv6 sets up the interrupt vector machinery
  lapic.c / lapicinit() -- tells LAPIC hardware to use vector 32 for timer
  trap.c / tvinit() -- initializes IDT, so entry i points to code at vector[i]
    this is mostly purely mechanical, IDT entries correspond blindly to vectors
    BUT T_SYSCALL's 1 (vs 0) tells CPU to leave interrupts enabled during system calls
    but not during device interrupts
    Q: why allow interrupts during system calls?
    Q: why disable interrupts during interrupt handling?
  vectors.S (generated by vectors.pl)
    first push fakes "error" slot in trapframe, since h/w doesn't push for some traps
    second push is just the vector number
      this shows up in trapframe as tf->trapno

how does the hardware know what stack to use for an interrupt?
  when it switches from user space to the kernel
  hardware-defined TSS (task state segment) lets kernel configure CPU
    one per CPU
    so each CPU can run a different process, take traps on different stacks
  proc.c / scheduler()
    one per CPU
  vm.c / switchuvm()
    tells CPU what kernel stack to use
    tells kernel what page table to use

Q: what eip should the CPU save when trapping to the kernel?
   eip of the instruction that was executing?
```

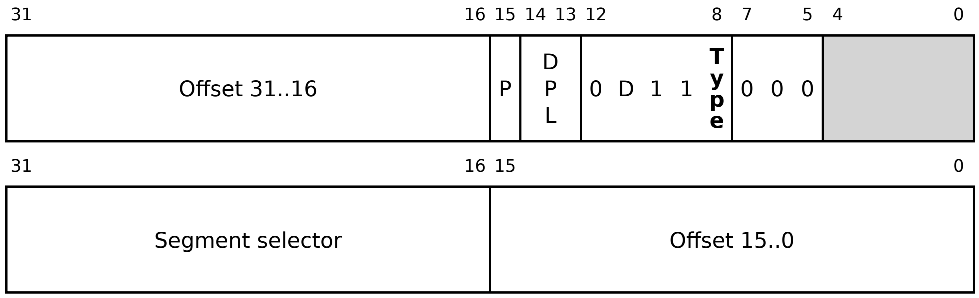
eip of the next instruction?
suppose the trap is a page fault?

some design notes

* interrupts used to be relatively fast; now they are slow
  old approach: every event causes an interrupt, simple h/w, smart s/w
  new approach: h/w completes lots of work before interrupting
* an interrupt takes on the order of a microsecond
  save/restore state
  cache misses
* some devices generate events faster than one per microsecond
  e.g. gigabit ethernet can deliver 1.5 million small packets / second
* polling rather than interrupting, for high-rate devices
  if events are always waiting, no need to keep alerting the software
* interrupt for low-rate devices, e.g. keyboard
  constant polling would waste CPU
* switch between polling and interrupting automatically
  interrupt when rate is low (and polling would waste CPU cycles)
  poll when rate is high (and interrupting would waste CPU cycles)
* faster forwarding of interrupts to user space
  for page faults and user-handled devices
  h/w delivers directly to user, w/o kernel intervention?
  faster forwarding path through kernel?

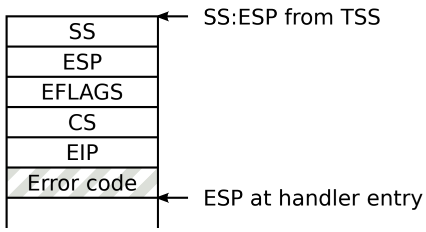
we will be seeing many of these topics later in the course

Interrupt/trap gate

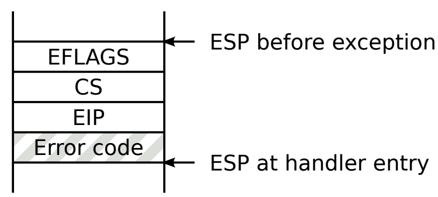


Type	0=Interrupt gate	P	Present
	1=Trap gate	DPL	Descriptor privilege level
Selector	Destination CS		(CPL required to invoke gate)
Offset	Destination IP or EIP	D	Size of gate (0=16-bits, 1=32-bits)

Exception stack (with privilege change)

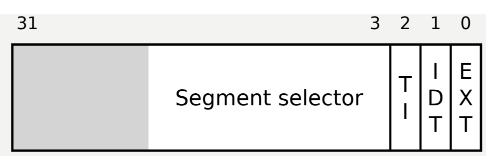


Exception stack (without privilege change)

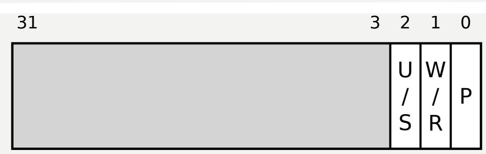


Vector	Description	Type	Error code
0	Divide error	Fault	No
1	Reserved		
2	Non-maskable interrupt	Interrupt	No
3	Breakpoint	Trap	No
4	Overflow	Trap	No
5	BOUND range exceeded	Fault	No
6	Invalid/undefined opcode	Fault	No
7	No math coprocessor	Fault	No
8	Double fault	Abort	Zero
9	Reserved		
10	Invalid TSS	Fault	Yes
11	Segment not present	Fault	Yes
12	Stack-segment fault	Fault	Yes
13	General protection	Fault	Yes
14	Page fault	Fault	Yes
15	Reserved		No
16	x87 FPU error	Fault	No
17	Alignment check	Fault	Zero
18	Machine check	Abort	No
19	SIMD FP exception	Fault	No
20-31	Reserved		
32-255	User defined interrupts	Interrupt	No

Exception types	
Fault	Faulting instruction not executed CS:EIP is the faulting instruction
Trap	Trapping instruction executed CS:EIP is the next instruction
Abort	Location is imprecise; cannot safely resume execution



TI	0=GDT, 1=LDT
IDT	0=GDT/LDT, 1=IDT
EXT	External event



P	0=Non-present page 1=Protection-violation
W/R	Cause (0=Read, 1=Write)
U/S	Mode (0=Supervisor, 1=User)

```
6.828 2016 : Locking

Why talk about locking?
  apps want to use multi-core processors for parallel speed-up
  so kernel must deal with parallel system calls
  and thus parallel access to kernel data (buffer cache, processes, &c)
  locks help with correct sharing of data
  locks can limit parallel speedup

Locking homework
  recall: ph.c, multi-threaded hash table, put(), get()
  vi ph0.c
  Q: why run ph.c on multiple cores?
     diagram: CPUs, bus, RAM
     assumption: CPU is bottleneck, can divide work between CPUs
  Q: can we beat single-core put() time? single-core get() time?
     ./ph0 1
  Q: how to measure parallel speedup?
     ./ph0 2
     twice as much work per unit time!
  Q: where are the missing keys?
  Q: specific scenario?
     diagram...
     table[0] = 15
     concurrent put(5), put(10)
     both insert()s allocate new entry, point next to 15
     both set table[0] to their new entry
     last inserter wins, other one is lost!
     called a "lost update"; example of a "race"
     race = concurrent accesses; at least one write
  Q: where to put the lock/unlock?
  Q: one lock covering the whole hash table?
     why? why not?
     called a "big" or "coarse-grained" lock
     ./ph1 2
     faster? slower? why?
  Q: one lock per table[] entry?
     this lock is "finer grained"
     why might this be good?
     ./ph2 2
     faster? slower? why?
     what might be harder with per-bucket locks?
     will we get a good speedup with 10 cores? NBUCKET=5...
  Q: one lock per struct entry, protecting the next pointer?
     why? why not?
  Q: does get() need to lock?
  Q: does get() need to lock if there are concurrent put()s?
     it's a race; but is it incorrect?

The lock abstraction:
  lock l
  acquire(l)
    x = x + 1 -- "critical section"
  release(l)
  a lock is itself an object
  if multiple threads call acquire(l)
    only one will return right away
    the others will wait for release() -- "block"
  a program typically has lots of data, lots of locks
    if different threads use different data,
    then they likely hold different locks,
    so they can execute in parallel -- get more work done.
  note that lock l is not specifically tied to data x
    the programmer has a plan for the correspondence

A conservative rule to decide when you need to lock:
  any time two threads use a memory location, and at least one is a write
  don't touch shared data unless you hold the right lock!
  (too strict: program logic may sometimes rule out sharing; lock-free)
```

```
  (too loose: printf(); not always simple lock/data correspondence)

Could locking be automatic?
  perhaps the language could associate a lock with every data object
    compiler adds acquire/release around every use
    less room for programmer to forget!
  that idea is often too rigid:
    rename("d1/x", "d2/y"):
      lock d1, erase x, unlock d1
      lock d2, add y, unlock d2
    problem: the file didn't exist for a while!
      rename() should be atomic
        other system calls should see before, or after, not in between
      otherwise too hard to write programs
    we need:
      lock d1 ; lock d2
      erase x, add y
      unlock d2; unlock d1
  that is, programmer often needs explicit control over
    the region of code during which a lock is held
    in order to hide awkward intermediate states

Ways to think about what locks achieve
  locks help avoid lost updates
  locks help you create atomic multi-step operations -- hide intermediate states
  locks help operations maintain invariants on a data structure
    assume the invariants are true at start of operation
    operation uses locks to hide temporary violation of invariants
    operation restores invariants before releasing locks

Problem: deadlock
  notice rename() held two locks
  what if:
    core A              core B
    rename(d1/x, d2/y)  rename(d2/a, d1/b)
      lock d1             lock d2
      lock d2 ...         lock d1 ...
  solution:
    programmer works out an order for all locks
    all code must acquire locks in that order
    i.e. predict locks, sort, acquire -- complex!

Locks versus modularity
  locks make it hard to hide details inside modules
  to avoid deadlock, I need to know locks acquired by functions I call
  and I may need to acquire them before calling, even if I don't use them
  i.e. locks are often not the private business of individual modules

How to think about where to place locks?
  here's a simple plan for new code
  1. write module to be correct under serial execution
     i.e. assuming one CPU, one thread
     insert(){ e->next = l; l = e; }
     but not correct if executed in parallel
  2. add locks to FORCE serial execution
     since acquire/release allows execution by only one CPU at a time
  the point:
    it's easier for programmers to reason about serial code
    locks can cause your serial code to be correct despite parallelism

What about performance?
  after all, we're probably locking as part of a plan to get parallel speedup

Locks and parallelism
  locks *prevent* parallel execution
  to get parallelism, you often need to split up data and locks
    in a way that lets each core use different data and different locks
    "fine grained locks"
  choosing best split of data/locks is a design challenge
```

```
    whole ph.c table; each table[] row; each entry
    whole FS; directory/file; disk block
    whole kernel; each subsystem; each object
  you may need to re-design code to make it work well in parallel
    example: break single free memory list into per-core free lists
      helps if threads were waiting a lot on lock for single free list
    such re-writes can require a lot of work!

Lock granularity advice
  start with big locks, e.g. one lock protecting entire module
    less deadlock since less opportunity to hold two locks
    less reasoning about invariants/atomicity required
  measure to see if there's a problem
    big locks are often enough -- maybe little time spent in that module
  re-design for fine-grained locking only if you have to

Let's look at locking in xv6.

A typical use of locks: ide.c
  typical of many O/S's device driver arrangements
  diagram:
    user processes, kernel, FS, iderw, append to disk queue
    IDE disk hardware
    ideintr
  sources of concurrency: processes, interrupt
  only one lock in ide.c: idelock -- fairly coarse-grained
  iderw() -- what does idelock protect?
    1. no races in idequeue operations
    2. if queue not empty, IDE h/w is executing head of queue
    3. no concurrent access to IDE registers
  ideintr() -- interrupt handler
    acquires lock -- might have to wait at interrupt level!
    uses idequeue (1)
    hands next queued request to IDE h/w (2)
    touches IDE h/w registers (3)

How to implement locks?
  why not:
    struct lock { int locked; }
    acquire(l) {
      while(1){
        if(l->locked == 0){ // A
          l->locked = 1;    // B
          return;
        }
      }
    }
  oops: race between lines A and B
  how can we do A and B atomically?

Atomic exchange instruction:
  mov $1, %eax
  xchg %eax, addr
  does this in hardware:
    lock addr globally (other cores cannot use it)
    temp = *addr
    *addr = %eax
    %eax = temp
    unlock addr
  x86 h/w provides a notion of locking a memory location
    different CPUs have had different implementations
    diagram: cores, bus, RAM, lock thing
    so we are really pushing the problem down to the hardware
    h/w implements at granularity of cache-line or entire bus
  memory lock forces concurrent xchg's to run one at a time, not interleaved

Now:
  acquire(l){
    while(1){
```

```
      if(xchg(&l->locked, 1) == 0){
        break
      }
    }
  }
  if l->locked was already 1, xchg sets to 1 (again), returns 1,
    and the loop continues to spin
  if l->locked was 0, at most one xchg will see the 0; it will set
    it to 1 and return 0; other xchgs will return 1
  this is a "spin lock", since waiting cores "spin" in acquire loop

Look at xv6 spinlock implementation
  spinlock.h -- you can see "locked" member of struct lock
  spinlock.c / acquire():
    see while-loop and xchg() call
    what is the pushcli() about?
      why disable interrupts?
  release():
    sets lk->locked = 0
    and re-enables interrupts

Detail: memory read/write ordering
  suppose two cores use a lock to guard a counter, x
  and we have a naive lock implementation
  Core A:          Core B:
    locked = 1
    x = x + 1      while(locked == 1)
    locked = 0       ...
                   locked = 1
                   x = x + 1
                   locked = 0
  the compiler AND the CPU re-order memory accesses
    i.e. they do not obey the source program's order of memory references
    e.g. the compiler might generate this code for core A:
      locked = 1
      locked = 0
      x = x + 1
      i.e. move the increment outside the critical section!
    the legal behaviors are called the "memory model"
  release()'s call to __sync_synchronize() prevents re-order
    compiler won't move a memory reference past a __sync_synchronize()
    and (may) issue "memory barrier" instruction to tell the CPU
  acquire()'s call to xchg() has a similar effect:
    intel promises not to re-order past xchg instruction
    some junk in x86.h xchg() tells C compiler not to delete or re-order
      (volatile asm says don't delete, "m" says no re-order)
  if you use locks, you don't need to understand the memory ordering rules
    you need them if you want to write exotic "lock-free" code

Why spin locks?
  don't they waste CPU while waiting?
  why not give up the CPU and switch to another process, let it run?
  what if holding thread needs to run; shouldn't waiting thread yield CPU?
  spin lock guidelines:
    hold spin locks for very short times
    don't yield CPU while holding a spin lock
  systems often provide "blocking" locks for longer critical sections
    waiting threads yield the CPU
    but overheads are typically higher
    you'll see some xv6 blocking schemes later

Advice:
  don't share if you don't have to
  start with a few coarse-grained locks
  instrument your code -- which locks are preventing parallelism?
  use fine-grained locks only as needed for parallel performance
  use an automated race detector
```

6.828 2017 Lecture 10: Processes, threads, and scheduling

Plan:
  homework
  thread switching
  scheduling

Homework

iderw():
  what does the lock protect?
  what goes wrong with adding sti/cli in iderw?
    iderw(): sti() after acquire, cli() before release
    let's try it...
  what would happen if acquire didn't check holding() and panic()?
    let's try it...
  what does happen to the interrupt, in the original code?
  what if IDE interrupt had occured on a different core?
  look at spinlock.c acquire/release to see interrupt manipulation
    why counting in pushcli/popcli?
    cpu->intena b/c syscalls run enabled, dev intrs run disabled
  why does acquire disable interrupts *before* waiting for the lock?

filealloc() in file.c, what if interrupts enabled?
  Q: what does ftable.lock protect?
  Q: why is there (usually) not a problem if interrupts enabled?
  Q: how might a problem nevertheless arise?
     try it: yield() after filealloc()'s acquire()

Process scheduling

Process
  an abstract virtual machine, as if it had its own CPU and memory,
    not accidentally affected by other processes.
  motivated by isolation

Process API:
  fork
  exec
  exit
  wait
  kill
  sbrk
  getpid

Challenge: more processes than processors
  your laptop has two processors
  you want to run three programs: window system, editor, compiler
  we need to multiplex N processors among M processes
  called time-sharing, scheduling, context switching

Goals for solution:
  Transparent to user processes
  Pre-emptive for user processes
  Pre-emptive for kernel, where convenient
    Helps keeps system responsive

xv6 solution:
  1 user thread and 1 kernel thread per process
  1 scheduler thread per processor
  n processors

What's a thread?
  a CPU core executing (with registers and stack), or
  a saved set of registers and a stack that could execute

Overview of xv6 processing switching
  user -> kernel thread (via system call or timer)
  kernel thread yields, due to pre-emption or waiting for I/O

```
  kernel thread -> scheduler thread
  scheduler thread finds a RUNNABLE kernel thread
  scheduler thread -> kernel thread
  kernel thread -> user

Each xv6 process has a proc->state
  RUNNING
  RUNNABLE
  SLEEPING
  ZOMBIE
  UNUSED

Note:
  xv6 has lots of kernel threads sharing the single kernel address space
  xv6 has only one user thread per process
  more serious O/S's (e.g. Linux) support multiple user threads per process

Context-switching was one of the hardest things to get right in xv6
  multi-core
  locking
  interrupts
  process termination

# Code

pre-emptive switch demonstration
  hog.c -- two CPU-bound processes
  my qemu has only one CPU
  let's look at how xv6 switches between them

timer interrupt
  run hog
  list trap.c:124
  breakpoint on yield()
  [stack diagram]
  print myproc()->name
  print myproc()->pid
  print/x tf->cs
  print/x tf->eip
  print tf->trapno (T_IRQ0+IRQ_TIMER = 32+0)
  step into yield
  state = RUNNABLE -- giving up CPU but want to run again
  step into sched

swtch -- to scheduler thread
  a context holds a non-executing kernel thread's saved registers
    xv6 contexts always live on the stack
    context pointer is effectively the saved esp
    (remember that *user* registers are in trapframe on stack)
    proc.h, struct context
  two arguments: from and to contexts
    push registers and save esp in *from
    load esp from to, pop registers, return
  confirm switching to scheduler()
    p/x *mycpu()->scheduler
    p/x &scheduler
  stepi -- and look at swtch.S
    [draw two stacks]
    eip (saved by call instruction)
    ebp, ebx, esi, edi
    save esp in *from
    load esp from to argument
    x/8x $esp
    pops
    ret
    where -- we're now on scheduler's stack

scheduler()
  print p->state
```

print p->name
print p->pid
swtch just returned from a *previous* scheduler->process switch
scheduler releases old page table, cpu->proc
 switchkvm() in vm.c
next a few times -- scheduler() finds other process
print p->pid
switchuvm() in vm.c
stepi through swtch()
 what's on the thread stack? context/callrecords/trapframe
 returning from timer interrupt to user space
 where shows trap/yield/sched

Q: what is the scheduling policy?
 will the thread that called yield() run immediately again?

Q: why does scheduler() release after loop, and re-acquire it immediately?
 To give other processors a chance to use the proc table
 Otherwise two cores and one process = deadlock

Q: why does the scheduler() briefly enable interrupts?
 There may be no RUNNABLE threads
  They may all be waiting for I/O, e.g. disk or console
 Enable interrupts so device has a chance to signal completion
  and thus wake up a thread

Q: why does yield() acquire ptable.lock, but scheduler() release it?
 unusual: the lock is released by a different thread than acquired it!
 why must the lock remain held across the swtch()?
 what if another core's scheduler() immediately saw the RUNNABLE process?

sched() and scheduler() are "co-routines"
 caller knows what it is swtch()ing to
 callee knows where switch is coming from
 e.g. yield() and scheduler() cooperate about ptable.lock
 different from ordinary thread switching, where neither
  party typically knows which thread comes before/after

Q: how do we know scheduler() thread is ready for us to swtch() into?
 could it be anywhere other than swtch()?

these are some invariants that ptable.lock protects:
 if RUNNING, processor registers hold the values (not in context)
 if RUNNABLE, context holds its saved registers
 if RUNNABLE, no processor is using its stack
 holding the lock from yield() all the way to scheduler enforces:
  interrupts off, so timer can't invalidate swtch save/restore
  another CPU can't execute until after stack switch

Q: is there pre-emptive scheduling of kernel threads?
 what if timer interrupt while executing in the kernel?
 what does kernel thread stack look like?

Q: why forbid locks from being held when yielding the CPU?
 (other than ptable.lock)
 i.e. sched() checks that ncli == 1
 an acquire may waste a lot of time spinnning
 worse: deadlock, since acquire waits with interrupts off

Thread clean up
 look at kill(pid)
  stops the target process
 can kill() free killed process's resources? memory, FDs, &c?
  too hard: it might be running, holding locks, etc.
  so a process must kill itself
  trap() checks p->killed
  and calls exit()
 look at exit()
  the killed process runs exit()

```
  can a thread free its own stack?
    no: it is using it, and needs it to swtch() to scheduler().
  so exit() sets proc->state = ZOMBIE; parent finishes cleanup
  ZOMBIE child is guaranteed *not* to be executing / using stack!
wait() does the final cleanup
  the parent is expected to call the wait() system call
  stack, pagetable, proc[] slot
```

```
6.828 2017 Lecture 11: Coordination (sleep&wakeup)

# reminder -- quiz next week
  this room, this time
  open notes, code, xv6 book, laptop (but no network!)

# plan
  finish scheduling
  user-level thread switch homework
  sequence coordination
    xv6: sleep & wakeup
    lost wakeup problem
    termination

# big picture:
  processes, kernel stack per process, cores, scheduling stack per core
  diagram of yield/swtch/scheduler/swtch/yield
    yield       scheduler
    acquire     release
    RUNNABLE
    swtch       swtch
      |           ^
      |           |
      -------------

# xv6's use of ptable.lock and swtch from kernel thread to scheduler is unusual
  for the most part xv6 is an ordinary parallel shared-memory program
  but this use of thread-switch and locks is very O/S-specific

# questions about xv6 context switch vs concurrency
  why does yield() hold ptable.lock across sched/swtch?
    what if another core's scheduler immediately saw RUNNABLE process?
    what if timer interrupt occured during swtch()?
  how do we know scheduler() thread is ready for us to swtch() into?
  can two scheduler()s select the same RUNNABLE process?

# homework: context switching for user-level threads
  show uthread_switch.S
  (gdb) symbol-file _uthread
  (gdb) b thread_switch
  (gdb) c
  uthread
  (gdb) p/x next_thread->sp
  (gdb) x/9x next_thread->sp
  Q: what's the 9th value on the stack?
     (gdb) p/x &mythread
  Q: why does the code copy next_thread to current_thread?
  Q: why OK for uthread yield to call scheduler, but not kernel?
  Q: what happens when a uthread blocks in a system call?
  Q: do our uthreads take advantage of multi-core for parallel execution?

# sequence coordination:
  threads need to wait for specific events or conditions:
    wait for disk read to complete
    wait for pipe reader(s) to make space in pipe
    wait for any child to exit

# why not just have a while-loop that spins until event happens?

# better solution: coordination primitives that yield the CPU
  sleep & wakeup (xv6)
  condition variables (homework)
  barriers (homework)
  etc.

# sleep & wakeup:
  sleep(chan, lock)
    sleeps on a "channel", an address to name the condition we are sleeping on
  wakeup(chan)
```

wakeup wakes up all threads sleeping on chan
may wake up more than one thread
no formal connection to the condition the sleeper is waiting for
and indeed sleep() can return even if condition isn't true
so caller must treat sleep() return as a hint

two problems arise in design of sleep/wakeup

- lost wakeup
- termination while sleeping

example use of sleep/wakeup -- iderw() / ideintr()

iderw() queues block read request, then sleep()s
b is a buffer which will be filled with the block content
iderw() sleeps waiting for B_VALID -- the disk read to complete
chan is b -- this buffer (there may be other processes in iderw())
ideintr() is called via interrupt when current disk read is done
marks b B_VALID
wakeup(b) -- same chan as sleep

puzzles
iderw() holds idelock when it calls sleep
but ideintr() needs to acquire idelock!
why doesn't iderw() release idelock before calling sleep()?
let's try it!

lost wakeup demo

modify iderw() to call release ; broken_sleep ; acquire
look at broken_sleep()
look at wakeup()
what happens?
ideintr() runs after iderw() saw no B_VALID
but before broken_sleep() sets state = SLEEPING
wakeup() scans proctable but no thread is SLEEPING
then sleep() sets current thread to SLEEPING and yields
sleep misses wakeup -> deadlock
this is a "lost wakeup"

xv6 lost wakeup solution:

goal:
1) lock out wakeup() for entire time between condition
check and state = SLEEPING
2) release the condition lock while asleep
xv6 strategy:
require wakeup() to hold both lock on condition AND ptable.lock
sleeper at all times holds one or the other lock
can release condition lock after it holds ptable.lock
look at ideintr()'s call to wakeup()
and wakeup itself
both locks are held while looking for sleepers
look at iderw()'s call to sleep()
condition lock is held when it calls sleep()
look at sleep() -- acquires ptable.lock, then releases lock on condition
diagram:

```
|----idelock----|
              |---ptable.lock---|
                                 |----idelock----|
                                  |-ptable.lock-|
```

thus:
either complete sleep() sequence runs, then wakeup(),
and the sleeper will be woken up
or wakeup() runs first, before potential sleeper checks condition,
but waker will have set condition true
requires that sleep() takes a lock argument

People have developed many sequence coordination primitives,

all of which have to solve the lost wakeup problem.
e.g. condition variables (similar to sleep/wakeup)
e.g. counting semaphores

```
# Another example: piperead()
  the while loop is waiting for more than zero bytes of data in buffer
    wakeup is at end of pipewrite()
    chan is &p->nread
  what is the race if piperead() used broken_sleep()?
  why is there a loop around the sleep()?
  why the wakeup() at the end of piperead()?

# second sequence coordination challenge -- how to terminate a sleeping thread?

# first, how does kill(target_pid) work?
  problem: may not be safe to forcibly terminate a process
    it might be executing in the kernel
      using its kernel stack, page table, proc[] entry
    might be in a critical section, needs to finish to restore invariants
    so we can't immediately terminate it
  solution: target exits at next convenient point
    kill() sets p->killed flag
    target thread checks p->killed in trap() and exit()s
      so kill() doesn't disturb e.g. critical sections
    exit() closes FDs, sets state=ZOMBIE, yields CPU
      why can't it free kernel stack and page table?
    parent wait() frees kernel stack, page table, and proc[] slot

# what if kill() target is sleep()ing?
  e.g. waiting for console input, or in wait(), or iderw()
  we'd like target to stop sleeping and exit()
  but that's not always reasonable
    maybe target is sleep()ing halfway through a complex operation
      that (for consistency) must complete
    e.g. creating a file

# xv6 solution
  see kill() in proc.c
    changes SLEEPING to RUNNABLE -- like wakeup()
  some sleep loops check for p->killed
    e.g. piperead()
    sleep() will return due to kill's state=RUNNABLE
    in a loop, so re-checks
      condition true -> reads some bytes, then trap ret calls exit()
      condition false -> sees p->killed, return, trap ret calls exit()
    either way, near-instant response to kill() of thread in piperead()
  some sleep loops don't check p->killed
    Q: why not modify iderw() to check p->killed in sleep loop and return?
    A: if reading, calling FS code expects to see data in the disk buffer!
       if writing (or reading), might be halfway through create()
       quitting now leaves on-disk FS inconsistent.
  so a thread in iderw() may continue executing for a while in the kernel
    trap() will exit() after the system call finishes

# xv6 spec for kill
  if target is in user space
    will die next time it makes a system call or takes a timer interrupt
  if target is in the kernel
    target will never execute another a user instruction
    but may spend quite a while yet in the kernel

# how does JOS cope with these problems?
  lost wakeups?
    JOS is uniprocessor and interrupts are disabled in the kernel
    so wakeup can't sneak between condition check and sleep
  termination while blocking?
    JOS has only a few system calls, and they are fairly simple
      no blocking multi-step operations like create()
      since no FS and no disk driver in the kernel
    really only one blocking call -- IPC recv()
      if env_destroy() is running, the target thread is not running
      recv() leaves env in a state where it can be safely destroyed
```

```
# Summary
  sleep/wakeup let threads wait for specific events
  concurrency and interrupts mean we have to worry about lost wakeups
  termination is a pain in threading systems
    context switch vs process exit
    sleeping vs kill
```

```
6.828 2017 Lecture 12: File System

lecture plan:
  file systems
  API -> disk layout
  caching

why are file systems useful?
  durability across restarts
  naming and organization
  sharing among programs and users

why interesting?
  crash recovery
  performance
  API design for sharing
  security for sharing
  abstraction is useful: pipes, devices, /proc, /afs, Plan 9
    so FS-oriented apps work with many kinds of objects
  you will implement one for JOS

API example -- UNIX/Posix/Linux/xv6/&c:
  fd = open("x/y", -);
  write(fd, "abc", 3);
  link("x/y", "x/z");
  unlink("x/y");

high-level choices visible in this API
  objects: files (vs virtual disk, DB)
  content: byte array (vs 80-byte records, BTree)
  naming: human-readable (vs object IDs)
  organization: name hierarchy
  synchronization: none (vs locking, versions)

a few implications of the API:
  fd refers to something
    that is preserved even if file name changes
    or if file is deleted while open!
  a file can have multiple links
    i.e. occur in multiple directories
    no one of those occurences is special
    so file must have info stored somewhere other than directory
  thus:
    FS records file info in an "inode" on disk
    FS refers to inode with i-number (internal version of FD)
    inode must have link count (tells us when to free)
    inode must have count of open FDs
    inode deallocation deferred until last link and FD are gone

let's talk about xv6

FS software layers
  system calls
  name ops | FD ops
  inodes
  inode cache
  log
  buffer cache
  ide driver

IDE is a standard to talk to devices
  https://en.wikipedia.org/wiki/Parallel_ATA
  connectors, interface, protocol, etc.
  CPU talks to IDE controller
  controller talks to hard drive, SSD, CD-ROM

hard disk drives (HDD)
  concentric tracks
  head must seek, disk must rotate
```

```
    random access is slow (5 or 10ms per access)
    sequential access is much faster (100 MB/second)
  each track is a sequence of sectors, usually 512 bytes
  ECC on each sector
  can only read/write whole sectors
  thus: sub-sector writes are expensive (read-modify-write)

solid state drives (SSD)
  NAND flash non-volatile memory
  random access: 100 microseconds
  sequential: 500 MB/second
  internally complex -- hidden except sometimes performance
    flash must be erased before it's re-written
    limit to the number of times a flash block can be written
    SSD copes with a level of indirection -- remapped blocks

for both HDD and SSD:
  sequential access is much faster than random
  big reads/writes are faster than small ones
  both of these have a big influence on high-performance FS design

disk blocks
  most o/s use blocks of multiple sectors, e.g. 4 KB blocks = 8 sectors
  to reduce book-keeping and seek overheads
  xv6 uses single-sector blocks for simplicity

on-disk layout
  xv6 file system on 2nd IDE drive; first has just kernel
  xv6 treats IDE drive as an array of sectors, ignores track structure
  0: unused
  1: super block (size, ninodes)
  2: log for transactions
  32: array of inodes, packed into blocks
  58: block in-use bitmap (0=free, 1=used)
  59: file/dir content blocks
  end of disk

"meta-data"
  everything on disk other than file content
  super block, i-nodes, bitmap, directory content

on-disk inode
  type (free, file, directory, device)
  nlink
  size
  addrs[12+1]

direct and indirect blocks

example:
  how to find file's byte 8000?
  logical block 15 = 8000 / 512
  3rd entry in the indirect block

each i-node has an i-number
  easy to turn i-number into inode
  inode is 64 bytes long
  byte address on disk: 2*512 + 64*inum

directory contents
  directory much like a file
    but user can't directly write
  content is array of dirents
  dirent:
    inum
    14-byte file name
  dirent is free if inum is zero

you should view FS as an on-disk data structure
```

```
  [tree: dirs, inodes, blocks]
  with two allocation pools: inodes and blocks

let's look at xv6 in action
  focus on disk writes
  illustrate on-disk data structures via how updated

Q: how does xv6 create a file?

rm fs.img

$ echo > a
write 34 ialloc (from create sysfile.c; mark it non-free)
write 34 iupdate (from create; initialize nlink &c)
write 59 writei (from dirlink fs.c, from create)

call graph:
  sys_open      sysfile.c
    create      sysfile.c
      ialloc    fs.c
      iupdate   fs.c
      dirlink   fs.c
        writei  fs.c

Q: what's in block 34?
   look at create() in sysfile.c

Q: why *two* writes to block 34?

Q: what is in block 59?

Q: what if there are concurrent calls to ialloc?
   will they get the same inode?
   note bread / write / brelse in ialloc
   bread locks the block, perhaps waiting, and reads from disk
   brelse unlocks the block

Q: how does xv6 write data to a file?

$ echo x > a
write 58 balloc  (from bmap, from writei)
write 508 bzero
write 508 writei (from filewrite file.c)
write 34 iupdate  (from writei)
write 508 writei
write 34 iupdate

call graph:
  sys_write       sysfile.c
    filewrite     file.c
      writei      fs.c
        bmap
          balloc
            bzero
        iupdate

Q: what's in block 58?
   look at writei call to bmap
   look at bmap call to balloc

Q: what's in block 508?

Q: why the iupdate?
   file length and addrs[]

Q: why *two* writei+iupdate?
   echo calls write() twice, 2nd time for the newline

Q: how does xv6 delete a file?
```

```
$ rm a
write 59 writei (from sys_unlink; directory content)
write 34 iupdate (from sys_unlink; link count of file)
write 58 bfree  (from itrunc, from iput)
write 34 iupdate (from itrunc; zeroed length)
write 34 iupdate (from iput; marked free)

call graph:
  sys_unlink
    writei
    iupdate
    iunlockput
      iput
        itrunc
          bfree
          iupdate
        iupdate

Q: what's in block 59?
   sys_unlink in sysfile.c

Q: what's in block 34?

Q: what's in block 58?
   look at iput

Q: why three iupdates?

Let's look at the block cache in bio.c
   block cache holds just a few recently-used blocks
   bcache at start of bio.c

FS calls bread, which calls bget
   bget looks to see if block already cached
   if present and not B_BUSY, return the block
   if present and B_BUSY, wait
   if not present, re-use an existing buffer
   b->refcnt++ prevents buf from being recycled while we're waiting

Two levels of locking here
  bcache.lock protects the description of what's in the cache
  buf->lock protects just the one buffer

Q: what is the block cache replacement policy?
   prev ... head ... next
   bget re-uses bcache.head.prev -- the "tail"
   brelse moves block to bcache.head.next

Q: is that the best replacement policy?

Q: what if lots of processes need to read the disk? who goes first?
   iderw appends to idequeue list
   ideintr calls idestart on head of idequeue list
   so FIFO

Q: is FIFO a good disk scheduling policy?
   priority to interactive programs?
   elevator sort?

Q: why does it make sense to have a double copy of I/O?
   disk to buffer cache
   buffer cache to user space
   can we fix it to get better performance?

Q: how much RAM should we dedicate to disk buffers?
```

6.828 2016 Lecture 13: Crash Recovery, Logging
=

```
Plan
  problem: crash recovery
    crash leads to inconsistent on-disk file system
    on-disk data structure has "dangling" pointers
  solutions:
    logging

Last xv6 lecture
  next week switch to papers
  quiz next week

Homework
  draw picture inode, double indirect, indirect, block
```

Why crash recovery

```
What is crash recovery?
  you're writing the file system
  then the power fails
  you reboot
  is your file system still useable?

the main problem:
  crash during multi-step operation
  leaves FS invariants violated
  can lead to ugly FS corruption

examples:
  create:
    new dirent
    allocate file inode
    crash: dirent points to free inode -- disaster!
    crash: inode not free but not used -- not so bad
  write:
    block content
    inode addrs[] and len
    indirect block
    block free bitmap
    crash: inode refers to free block -- disaster!
    crash: block not free but not used -- not so bad
  unlink:
    block free bitmaps
    free inode
    erase dirent

what can we hope for?
  after rebooting and running recovery code
  1. FS internal invariants maintained
     e.g., no block is both in free list and in a file
  2. all but last few operations preserved on disk
     e.g., data I wrote yesterday are preserved
     user might have to check last few operations
  3. no order anomalies
     echo 99 > result ; echo done > status

simplifying assumption: disk is fail-stop
  disk executes the writes FS sends it, and does nothing else
    perhaps doesn't perform the very last write
  thus:
    no wild writes
    no decay of sectors

correctness and performance often conflict
  safety => write to disk ASAP
  speed => don't write the disk (batch, write-back cache, sort by track, &c)
```

Logging solution

most popular solution: logging (== journaling)
  goal: atomic system calls w.r.t. crashes
  goal: fast recovery (no hour-long fsck)

will introduce logging in two steps
  first xv6's log, which only provides safety and fast recovery
  then Linux EXT3, which is also fast normal operation

the basic idea behind logging
  you want atomicity: all of a system call's writes, or none
    let's call an atomic operation a "transaction"
  record all writes the sys call *will* do in the log (log)
  then record "done" (commit)
  then do the writes (install)
  on crash+recovery:
    if "done" in log, replay all writes in log
    if no "done", ignore log
  this is a WRITE-AHEAD LOG

challenge: avoid cache eviction
  cannot write dirty buffer back to their home location
  it would break the atomicity of transaction
  consider create example:
        write dirty inode to log
        write dir block to log
        evict dirty inode
        commit
        Q: will we recover correctly
  solution: pin dirty blocks in buffer cache
  after install, unpin block

xv6 log representation
  [diagram: buffer cache, in-memory log, FS tree on disk, log on disk]
  on write add blockno to in-memory array
  keep the data itself in buffer cache (pinned)
  on commit, write buffers to the log on disk
     could write log in one batch
  (after commit
        write the buffers in the log to their home location)

challenge: system's call data must fit in log
  compute an upperbound of number of blocks each calls writes
    set log size >= upper bound
  break up some system calls in several transactions
    large writes

challenge: allowing concurrent system calls
  must allow writes from several calls to be in log
  on commit must write them all
    to maintain order between sys calls
  BUT cannot write data from calls still in a transaction

xv6 solution
  install log when no systems calls are in a transaction
  count number of calls in system calls
  allow no new system calls to start if their data might not fit in log
    we computed an upper bound of number of blocks each calls writes
    if sys call doesn't fit, block its thread and wait until log has been installed
        (nice that each user-level thread has its own kernel thread)
  note: give up on some immediate durability
    when system call returns, data may not be on disk
        not a real problem: real file systems trade immediate durability for performance

challenge: overwrite the same block several times
  should we remove old block from log?
  append we new one to end of log?
  not really necessary

```
    ordering isn't important
        they will be applied as a single group
  thus, it is ok to absorb the write
    if block is already in log, don't do anything
```

Let's look at an example:

```
        $ echo a > x

        // transaction 1: create
        write 3
        write 4
        write 2
        write 34
        write 59
        write 2

        // transaction 2: write
        write 3
        write 4
        write 5
        write 2
        write 58
        write 565
        write 34
        write 2

        // transaction 3: write
        write 3
        write 4
        write 2
        write 565
        write 34
        write 2
```

let's look at filewrite (2nd trans)

```
    begin_op()
      bp = bread()
      bp->data[] = ...
      log_write(bp)
      more writes ...
    end_op()

   compute how max blocks we can write before log is full
   write that max blocks in a transaction

  `begin_op()`:
    need to indicate which group of writes must be atomic!
    need to check if log is being committed
    need to check if our writes will fit in log
        `begin_op` before ilock to avoid deadlock
  `log_write()`:
    record sector # in in-memory log
    don't append buffer sector content to log, but leave in buffer cache
    will set `B_DIRTY`, so that block won't be evicted
       see bio.c
  `end_op()`:
    if no outstanding operations, commit
  `commit()`:
    put in-memory log onto disk
      copy data from buffer cache into log
    record "done" and sector #s in log
    install writes from log into home location
      second disk write
      ide.c will clear B_DIRTY for block written --- now it can be evicted
    erase "done" from log
```

What would have happened if we crashed during a transaction?

```
  `recover_from_log()` is called on boot
   if log says "done":
      copy blocks from log to real locations on disk

how to set MAXOPBLOCKS and LOGSIZE?
  MAXOPBLOCK = 10
    create
  LOGSIZE = 3 * MAXOPBLOCKS
    some concurrency

what needs to be wrapped in transactions?
  many obvious examples (e.g., example above)
  but also less obvious ones:
    iput()
        namei()
  => everything that might update disk

concrete example why iput() should be wrapped:
 don't wrap iput in sys_chdir()
 $ mkdir abc
 $ cd abc
 $ ../rm ../abc
 $ cd ..
 It will cause a panic("write outside of trans");
 iput() might write when refcnt becomes 0

what is good about this design?
  correctness due to write-ahead log
  good disk throughput: log naturally batches writes
    but data disk blocks are written twice
  concurrency

what's wrong with xv6's logging?
  log traffic will be huge: every operation is many records
  logs whole blocks even if only a few bytes written
    worse, xv6 reads a block from disk even when it will be overwritten completely
        see homework for next lecture
  each block in log is written synchronously in write_log()
    could give write them as a batch and only write head synchronously
  eager write to real location -- slow
    could delay writes until log must be flushed (i.e, group commit)
  every block written twice
  trouble with operations that don't fit in the log
    unlink might dirty many blocks while truncating file
```

6.828 2016 Lecture 14: Linux ext3 crash recovery
=

```
Plan
  logging for cash recovery
    xv6: slow and immediately durable
    ext3: fast but not immediately durable
  trade-off: performance vs.safety

example problem:
  appending to a file
  two writes:
    mark block non-free in bitmap
    add block # to inode addrs[] array
  we want atomicity: both or neither
  so we cannot do them one at a time

why logging?
 atomic system calls w.r.t. crashes
 fast recovery (no hour-long fsck)
```

xv6
--

```
review of xv6 logging
  [diagram: buffer cache, in-memory log, FS tree on disk, log on disk]
  in-memory log: blocks that must be appended to log, in order
  log "header" block and data blocks
  each system call is a transaction
    begin_op, end_op
  syscall writes in buffer cache and appends to in-memory log
    some opportunity for write absorption
  at end_op, each written block appended to on-disk log
    but NOT yet written to "home" location
    "write-ahead log"
    preserve old copy until sure we can commit
  on commit:
    write "done" and block #s to header block
  then write modified blocks to home locations
  then erase "done" from header blocks
  recovery:
    if log says "done":
      copy blocks from log to real locations on disk

homework
   Q: what does "cat a" produce after panic?
          cannot open a
   Q: how about "ls"
           panic unknown inode type
   Problem:
     dirent is on disk and has an inode#
         that inode hasn't been written to disk to the right place
         in fact, on-disk it is marked as free
   Q: what does "cat a" produce after recovery?
     empty file.
         recovery wrote inode to the right place
           it is now allocated
           dirent is valid
     create and write are separate transactions
         create made it but write didn't
   modification to avoid reading log in install_trans()
         why is buffer 3 still in buffer cache?

what's wrong with xv6's logging? it is slow!
  all file system operation results in commit
        if no concurrent file operations
  synchronous write to on-disk log
    each write takes one disk rotation time
    commit takes a another
```

```
    a file create/delete involves around 10 writes
    thus 100 ms per create/delete -- very slow!
  tiny update -> whole block write
    creating a file only dirties a few dozen bytes
    but produces many kilobytes of log writes
  synchronous writes to home locations after commit
    i.e. write-through, not write-back
    makes poor use of in-memory disk cache

Ext3
--

how can we get both performance and safety?
  we'd like system calls to proceed at in-memory speeds
  sol: use write-back disk cache


write-back cache
  *no* sync meta-data update
  operations *only* modify in-memory disk cache (no disk write)
    so creat(), unlink(), write() &c return almost immediately
  bufs written to disk later
    if cache is full, write LRU dirty block
    write all dirty blocks every 30 seconds, to limit loss if crash
  this is how old Linux EXT2 file system worked

would write-back cache improve performance? why, exactly?
  after all, you have to write the disk in the end anyway
  typical system call complete w/o actual disk writes
  can do I/O concurrently with system calls

write-back cache: trades immediate durability for performance
  after system call returns modifications are not on disk
  give applications control of when to flush: sync, fsync()

what can go wrong w/ write-back cache?
  example: unlink() followed by create()
    an existing file x with some content, all safely on disk
    one user runs unlink(x)
      1. delete x's dir entry **
      2. put blocks in free bitmap
      3. mark x's inode free
    another user then runs create(y)
      4. allocate a free inode
      5. initialize the inode to be in-use and zero-length
      6. create y's directory entry **
    again, all writes initially just to disk buffer cache
    suppose only ** writes forced to disk, then crash
    what is the problem?

Linux's ext3 design
 case study of the details required to add logging to a file system
 Stephen Tweedie 2000 talk transcript "EXT3, Journaling Filesystem"
        http://olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html
 ext3 adds a log to ext2, a previous xv6-like log-less file system
 has many modes, I'll start with "journaled data"
   log contains both metadata and file content blocks

ext3 structures:
 in-memory write-back block cache
 in-memory list of blocks to be logged, per-handle
 on-disk FS
 on-disk circular log file.

what's in the ext3 log?
 superblock: starting offset and starting seq #
 descriptor blocks: magic, seq, block #s
 data blocks (as described by descriptor)
 commit blocks: magic, seq
```

```
 |super: offset+seq #|...|Descriptor 4+magic|...metadata blocks...|Commit 4+magic| |Descriptor 5+magic|...

how does ext3 get good performance despite logging entire blocks?
 batches many syscalls per commit
 defers copying cache block to log until it commits log to disk
 hopes multiple sycalls modified same block
   thus many syscalls, but only one copy of block in log
   much more "write absorbtion" than xv6

sys call:
  h = start()
  get(h, block #)
    warn logging system we'll modify cached block
      added to list of blocks to be logged
    prevent writing block to disk until after xaction commits
  modify the blocks in the cache
  stop(h)
    guarantee: all or none
    stop() does *not* cause a commit
  notice that it's pretty easy to add log calls to existing code

ext3 transaction
  [circle set of cache blocks in this xaction]
  while "open", adds new syscall handles, and remembers their block #s
  only one open transaction at a time
  ext3 commits current transaction every few seconds (or on sync()/fsync())

committing a transaction to disk
  open a new transaction, for subsequent syscalls
  mark transaction as done
  wait for in-progress syscalls to stop()
    (maybe it starts writing blocks, then waits, then writes again if needed)
  write descriptor to log on disk w/ list of block #s
  write each block from cache to log on disk
  -> wait for all log writes to finish
  append the commit record
  -> wait until commit record is on disk
  now cached blocks allowed to go to homes on disk (but not forced)
  cost: two write barriers

is log correct if concurrent syscalls?
  e.g. create of "a" and "b" in same directory
  inode lock prevents race when updating directory
  other stuff can be truly concurrent (touches different blocks in cache)
  transaction combines updates of both system calls

what if syscall B reads uncommited result of syscall A?
  A: echo hi > x
  B: ls > y
  could B commit before A, so that crash would reveal anomaly?
  case 1: both in same xaction -- ok, both or neither
  case 2: A in T1, B in T2 -- ok, A must commit first
  case 3: B in T1, A in T2
    could B see A's modification?
    ext3 must wait for all ops in prev xaction to finish
      before letting any in next start
      so that ops in old xaction don't read modifications of next xaction

T2 starts while T1 is committing to log on disk
  what if syscall in T2 wants to write block in prev xaction?
  can't be allowed to write buffer that T1 is writing to disk
    then new syscall's write would be part of T1
    crash after T1 commit, before T2, would expose update
  T2 gets a separate copy of the block to modify
    T1 holds onto old copy to write to log
  are there now *two* versions of the block in the buffer cache?
    no, only the new one is in the buffer cache, the old one isn't
  does old copy need to be written to FS on disk?
    no: T2 will write it
```

when can ext3 free a transaction's log space?
  after cached blocks have been written to FS on disk
  free == advance log superblock's start pointer/seq

what if block in T1 has been dirtied in cache by T2?
  can't write that block to FS on disk
  note ext3 only does copy-on-write while T1 is commiting
    after T1 commit, T2 dirties only block copy in cache
  so can't free T1 until T2 commits, so block is in log
    T2's logged block contains T1's changes

what if not enough free space in log for a syscall?
  suppose we start adding syscall's blocks to T2
  half way through, realize T2 won't fit on disk
  we cannot commit T2, since syscall not done
  can we free T1 to free up log space?
  maybe not, due to previous issue, T2 maybe dirtied a block in T1
  deadlock!

solution: reservations
  syscall pre-declares how many block of log space it might need
  block the sycall from starting until enough free space
  may need to commit open transaction, then free older transaction
    OK since reservations mean all started sys calls can complete + commit

what if a crash?
  crash may interrupt writing last xaction to log on disk
  so disk may have a bunch of full xactions, then maybe one partial
  may also have written some of block cache to disk
    but only for fully committed xactions, not partial last one

how does recovery work
  done by e2fsck, a utility program
  1. find the start and end of the log
     log "superblock" at start of log file
     log superblock has start offset and seq# of first transaction
     scan until bad record or not the expected seq #
     go back to last commit record
     crash during commit -> last transaction ignored during recovery
  2. replay all blocks through last complete xaction, in log order

what if block after last valid log block looks like a log descriptor?
  perhaps left over from previous use of log? (seq...)
  perhaps some file data happens to look like a descriptor? (magic #...)

performance?
  create 100 small files in a directory
    would take xv6 over 10 seconds (many disk writes per syscall)
  repeated mods to same direntry, inode, bitmap blocks in cache
    write absorbtion...
  then one commit of a few metadata blocks plus 100 file blocks
  how long to do a commit?
    seq write of 100*4096 at 50 MB/sec: 10 ms
    wait for disk to say writes are on disk
    then write the commit record
    that wastes one revolution, another 10 ms
    modern disk interfaces can avoid wasted revolution

ext3 not as immediately durable as xv6
  creat() returns -> maybe data is not on disk! crash will undo it.
  need fsync(fd) to force commit of current transaction, and wait
  would ext3 have good performance if commit after every sys call?
    would log fewer blocks, less absorption
    10 ms per syscall, rather than 0 ms
  (Rethink the Sync addresses this problem)

ordered vs data journaled mode

```
  journaling file content is slow, every data block written twice
  perhaps not needed to keep FS internally consistent
  can we just lazily write file content blocks?
  no:
    if metadata updated first, crash may leave file pointing
    to blocks with someone else's data
  ext3 ordered mode:
        by pass log for content blocks
    write content block to disk before commiting inode w/ new block #
    thus won't see stale data if there's a crash
  if crash before commit:
        file has new data
        but no metadata inconsistencies
  most people use ext3 ordered mode

correctness challenges w/ ordered mode:
  A. rmdir, re-use block for file, ordered write of file,
       crash before rmdir or write committed
     now scribbled over the directory block
     fix: defer free of block until freeing operation forced to log on disk
  B. rmdir, commit, re-use block in file, ordered file write, commit,
       crash, replay rmdir
     file is left w/ directory content e.g. . and ..
     fix: revoke records, prevent log replay of a given block

Summary of rules
  Don't write block to file system until committed
  Wait for all ops in T1 to finish before starting T2
  Don't overwrite a block in buffer cache before it is in the log
  Ordered mode:
        Write datablock to fs before commit
    Don't reuse free block until after free op committed
    Don't replay revoked operations

another corner case: open fd and unlink
  open a file, then unlink it
  unlink commits
  file is open, so unlink removes dir entry but doesn't free blocks
  crash
  nothing interesting in log to replay
  inode and blocks not on free list, also not reachably by any name
    will never be freed! oops
  solution: add inode to linked list starting from FS superblock
    commit that along with remove of dir ent
  recovery looks at that list, completes deletions

checksums to avoid one write barrier
  can disk barrier after writing data be avoided?
        write all data plus commit block
  risk: disks usually have write caches and re-order writes, for performance
    sometimes hard to turn off (the disk lies)
    people often leave re-ordering enabled for speed, out of ignorance
  bad news if disk writes commit block before preceding stuff
  solution: commit block contains checksum of all data blocks
    on recovery: compute checksum of datablocks
        if matche checksum in commit block: install transaction
    if no match: don't install transactions
  ext4 has journal checksumming

ext4 correctness challenge w. ordered mode+checksums
  write metadata to log, write content to on-disk, write commit, then barrier
    disk can re-order
  what if crash before barrier:
    content-blocks may not have been written
    log blocks and commit block may have been written
        replay log
        inodes may point to disk blocks with old content
        oops --- this bug was discovered 6 years after introduction
  sol: ext4 forbids ordered mode and checksums
```

does ext3 fix the xv6 log performance problems?
  synchronous write to on-disk log -- yes, but 5-second window
  tiny update -> whole block write -- yes (indirectly)
  synchronous writes to home locations after commit -- yes
  ext3/ext4 very successful

Notes on Ext4
--

* Ordered mode

In ordered mode ext4 flushes all dirty data of a file directly to the main file system, but it orders those updates before it flushes the corresponding metadata to the on-disk log. Ordered-mode ensures, for example, that inodes will always be pointing to valid blocks. But, it has the (perhaps undesirable) feature that data of a file may have been updated on disk, but the file's metadata may not have been updated.

Note that "ordered" mode doesn't refer some order among file operations such as create, read, write, etc.. "Ordered" refers to that dirty data blocks are written before the corresponding metadata blocks.

* Implementation

There is a file system (e.g., ext4) and a logging system (e.g., jdb2). Ext4 uses jbd2 to provide logging.

jbd2 maintains an in-memory log and an on-disk log. It provides handles to allow file systems to describe atomic operations. It may group several handles into a single transaction.

Ext4 uses jbd2 as follows. Systems calls in ext4 are made atomic using jbd2's handles. A system call starts with start_handle(). All blocks updated by that system call are associated with that handle. Each block has a buffered head, which describes that block. It may also have a journal head too, if jbd2 knows about it, which records the handle of the block, whether it contains metadata, and so on. When a ext4 opens a handle that handle is appended to list of handles for the current open transaction (and reserves sufficient space in the log). Ext4 marks the super block, blocks containing inodes, directory blocks, etc. as metadata blocks.

jbd2 maintains the in-memory log implicitly: it is the list of blocks associated with each handle in this transaction. jbd2 maintains a list of metadata blocks. jbd2 also maintains a list of dirty inodes that are part of this transaction.

When committing the in-memory log to the on-disk log, jbd2 flushes first the dirty, non-metadata blocks to their final destination (i.e., without going through the log). It does so by traversing the list of inodes in this transaction, and for each block of an inode it flushes it if it is dirty and not marked as metadata blocks. Then, jbd2 syncs the metadata blocks to the on-disk log.

Ext4+jbd2 guarantee a prefix property: if a file system operation x finishes before file system operation y starts, then x will be on disk before y. If x and y are closes to each other in time, they may be committed in the same transaction (which is fine). Y may also end up in a later transaction (which is fine). But, y will never end up in an earlier transaction than x.

When calling fsync(), ext4 waits for the current transaction to commit, which flushes the in-memory log to disk. Thus, fsync guarantees the prefix property for metadata operations: the metadata operations preceding fsync() are on disk when the fsync() completes.

Can a system call y observe results from some other call x by another process in memory and be ordered before x in the on-disk log? It is up to the file system and its concurrency mechanism, but for ext4 the answer is no. If y starts after x completes, the answer is definitely no (because of prefix property). If two calls run concurrently, then both will be committed to the disk in the same transaction, because once ext4 opens a handle, it is guaranteed to be part of the current transaction. on a commit, ext4 waits until all current active handles are closed before committing.

When opening a handle, ext4 must say how many blocks it needs in the log to complete the handle so that jbd2 can guarantee that all active handles can be committed in the current transaction. If there isn't enough space, then the start handle will be delayed until the next transaction.

* Implication for applications

An app doesn't have to sync a newly-created parent directory when fsyncing a file in that directory (assuming that the handle for the parent directory is ordered before the handle of the file). If ext4 orders handles correctly, jbd2 will write them to the on-disk log in the order of their handles.

(Note: if ext4 is run without a log, the code explicitly checks for this case, and forces a flush on the parent directory when a file in that directory is fsynced, see ext4/sync.c)

Why does the alice paper have an X for ext4-ordered [append -> any op]? the append updates metadata so the dirty block should be flushed before the metadata changes. maybe this has to do with delayed block allocation?

VM primitives for user programs by Appel & Li

What's the point?
  O/S should provide better support for user-controlled VM.
    Faster. More correct. More complete.
  Would make programs faster.
  Would allow neat tricks that are otherwise too painful.
  They provide laundry list of examples of uses.
  They analyze O/S VM efficiency, argue plenty of room for improvement.
  Do they define a new VM interface or design or implementation?

What are the primitives?
  TRAP, PROT1, PROTN, UNPROT, DIRTY, MAP2
  Are any of these hard? (MAP2...)
  Are any not easy in a simple (VAX-like) VM model?

What does PROTx actually do?
  Mark PTE/TLB "protected".
  And/or mark O/S vm structures "protected".
    And at least invalidate h/w PTE/TLB.
  Make sure it's not going to look like a page fault for disk paging...

What does TRAP actually mean?
  PTE (or TLB entry) marked "protected"
  CPU saves user state, jumps into kernel.
  Kernel asks VM system what to do?
    I.e. page in from disk? Core dump?
  Generate signal -- upcall into user process.
    Lower on user stack now, or on separate stack...
  Run user handler, can do anything.
    Probably must call UNPROT for referenced page.
    That is, must avoid repeated fault.
    Maybe we can change faulting address/register???? Maybe not.
  User handler returns to kernel.
  Kernel returns to user program.
  Continue or re-start instruction that trapped.

Were the primitives available in 1991 O/S's?

Were the primitives fast?
  What would fast mean?
    Perhaps relative to compiler-generated checking code?
    Perhaps relative to what we were going to do to handle the fault?
  Are they faster today?
    (needs to be relative to ordinary instruction times)
    12 microseconds on 1.2 GHz Athlon, FreeBSD 4.3. For trap, unprot, prot.

Do we really need VM hardware for these primitives?
  Not a security issue, so can be user controlled.
  Why doesn't RISC ideology apply?
  Why not have cc (or Atom) generate code to simulate VM?
    More flexible...
    Might be as fast; spare execution units.
    But it's a pain to modify the compiler (hence Atom).
  CPUs already have VM h/w, why not use it?
    Because then the O/S has to be involved. And it's slow and rigid.
    Cheap embedded CPUs don't have VM.
  For ordinary people, much easier to use VM than hack the compiler.

Let's look at concurrent GC.
  1. How does two-space compacting copying GC work?
     Need forwarding pointers in old space (and "copied" flag).
     Why is this attractive? Alloc is cheap. Compacts, so no free list.
     Why isn't it perfect?
  2. How does Baker's incremental GC work?
     Especially "scanned area" of to-space.
     Every load from non-scanned to-space must be checked.
       Does it point back to from-space?
     Must leave forwarding pointers in from-space for copied objects.

Incremental: every allocation scans a little.
3. How does VM help?
   Avoid explicit checks for ptrs back to from space.
     By read-protecting unscanned area.
     Why can't we just read-protect from-space?
   Also, a concurrent collector on another CPU.
     Why no conflict?
     Collector only reads from-space and protected unscanned to-space.
     Need sync when mutator thread traps.

Are existing VM primitives good enough for concurrent GC?
  MAP2 is the only functionality issue -- but not really.
    We never have to make the same page accessible twice!
  Are traps &c fast enough?
    They say no: 500 us to scan a page, 1200 us to take the trap.
    Why not scan 3 pages?
    How much slower to run Baker's actual algorithm, w/ checks?
      VM version might be faster! Even w/ slow traps.
    What about time saved by 2nd CPU scanning? They don't count this.
  Is it an issue how often faults occur for concurrent GC?
    Not really -- more faults means more scanning.
    I.e. we'll get <= one fault per page, at most.

```
6.828 2016 L15: Operating System Organization

Topic: what should a kernel do?
  What kinds of system calls should it support?
  What abstractions should it provide?

These are subjective questions!
  There are no definite answers
  But plenty of ideas, opinions, and debates
    We'll see some in the papers over the rest of the term
  This topic is more about ideas and less about specific mechanisms

The traditional approach
  1) big abstractions, and
  2) a "monolithic" kernel implementation
  UNIX, Linux, xv6
  kernel is a single big program
  kernel provides many powerful abstractions
  kernel hides much detail from applications
  clean interfaces, sophisticated implementations

Example: traditional treatment of CPU
  as if process (or thread) had its own dedicated CPU
    a "virtual" CPU
  very convenient:
    a process executes strictly sequentially, no suprises
    no need to worry about other processes
  implications:
    interrupts must save/restore all registers for transparency
    timer interrupts force transparent context switches
    system calls look like function calls: they return, perhaps after blocking
    process must ask for input, e.g. with read() or ipc_recv()
    process can't have much opinion about detailed scheduling

Example: traditional treatment of memory
  each process has its own private memory
    "address space" by virtualizing memory
    uniform array of storage locations
  very convenient:
    no need to worry about other process's use of memory
    no need to worry about security, since private
    kernel has great freedom to play clever virtual memory tricks
  implication:
    limited scope for programs to do their own clever VM tricks

Clever virtual memory tricks played by monolithic kernels:
  copy-on-write fork() (like Lab 4 but hidden in the kernel)
  demand paging:
    process bigger than available physical memory?
    "page-out" (write) pages to disk, mark PTEs invalid
    if process tries to use one of those pages, MMU causes page fault
      kern finds phys mem, page-in from disk, mark PTE valid
      then return to process -- transparent
    this works because apps use only a fraction of mem at a given time
  shared physical memory for executables and libraries
  fast I/O by having read()/write() borrow pages rather than copying
  paging h/w has turned out to be one of the most fruitful ideas in OS!
    due to the level of indirection, ability to intercept memory references
  would be nice if applications could implement their own similar tricks...

The intellectual habits of successful traditional kernels:
  portable interfaces
    files, not disk controller registers
    address spaces, not TLB or page table
  clean interfaces, hidden complexity
    all I/O via FDs and read/write, not specialized for each device &c
    address spaces with transparent disk paging
  big abstractions that give kernel scope to manage resources
    implement sophisticated mgmt once, in kernel, rather than in every app
```

    process abstraction lets kernel be in charge of scheduling
    file/directory abstraction lets kernel be in charge of disk layout
  big abstractions that allow kernel to understand and enforce security
    file system permissions
    processes with private address spaces
    (this is a big deal -- "kernel must implement to prevent buggy/evil programs from abusing")
  lots of indirection
    e.g. FDs, virtual addresses, file names, PIDs
    helps kernel virtualize, revoke, schedule, &c

The driving force behind big abstractions: app developers want convenience
  app developers want to spend time building new application features
  they want the O/S to deal with everything else
  so they want power and portability and reasonable speed

Monolithic implementation popular for O/Ss with big abstractions
  easy for sub-systems to cooperate -- no irritating boundaries
    e.g. integrated paging and file system cache
  all code runs with high privilege -- no internal security restrictions
  particularly attractive if O/S provides lots of big abstractions

What are the main criticisms of traditional kernels?
  big => complex, buggy, unreliable (in principle, not so much in practice)
  powerful abstractions tend to be over-general and thus slow
    maybe I don't need all my registers saved on a context switch
  abstractions are sometimes not quite right
    maybe I want to wait for a process that's not my child
  abstractions can prevent low-level optimizations
    DB may be better at laying out B-Tree files on disk than O/S FS

Microkernels -- an alternate approach
  big idea: move most O/S functionality to user-space service processes
  kernel can be small, mostly IPC
  [diagram: h/w, kernel, services (FS VM net), apps]
  the hope:
    simple kernel can be fast and reliable
    services are easier to replace and customize
  JOS is a mix of micro-kernel and exokernel

Microkernel wins:
  you really can make IPC fast
  services can sometimes be isolated for reliability
  maybe services are easy to distribute or run in parallel
  services force kernel developers to think about modularity
    e.g. lots of work done on virtual memory services

Microkernel losses:
  kernel can't be tiny: needs to know about processes and memory
  you may do lots of IPCs, slow in aggregate
  cross-sub-system optimization harder
  hard to be modular, can lead to a few huge services, not a big win
  hard to survive the crash of an important service

Microkernels have seen some success
  IPC/service idea widely used in e.g. OSX (which is not at all micro)
  some embedded O/Ss have strong microkernel flavor
  More next lecture

Exokernel (1995)

the paper:
  O/S community paid lots of attention
  full of interesting ideas
  describes an early research prototype
  not a complete system or complete explanation
    later SOSP 1997 paper realizes more of the vision

Exokernel overview
  philosophy: eliminate all abstractions

    for any problem, expose h/w or info to app, let app do what it wants
    a search for the minimum kernel functionality
    a theory that moving most functionality to apps will be possible and beneficial
  h/w, kernel, environments, libOS, app
  an exokernel would not provide address space, virtual cpu, pipes, file system, TCP
  instead, give control to app:
    phys pages, MMU mappings, clock interrupts, disk i/o, net i/o
    let app or libOS build nice address space if it wants, or not
    should give aggressive apps much more flexibility
  challenges:
    what interfaces will allow most of OS to be moved to libraries?
    can kernel defend against buggy/malicious library OSs?
    can you get sharing and security among multiple library OSs?
    can you get good performance despite more fine-grained syscalls?
    will there be a net benefit after all the effort?

Exokernel memory interface
  what are the resources? (phys pages, mappings)
  what does an app need to ask the kernel to do?
    pa = AllocPage()
    DeallocPage(pa)
    TLBwr(va, pa)
    Grant(env, pa)
  and these kernel->app upcalls:
    PageFault(va)
    PleaseReleaseAPage()
  what does exokernel need to do?
    track what env owns what phys pages
    ensure app only creates mappings to phys pages it owns
    decide which app to ask to give up a phys page when system runs out
      that app gets to decide which of its pages
  are there security problems?
  are there efficiency problems?

shared memory example
  two processes want to share memory, for fast interaction
    note traditional "virtual address space" doesn't allow for this
  process a: pa = AllocPage()
             put 0x5000 -> pa in private table
             PageFault(0x5000) upcall -> TLBwr(0x5000, pa)
             Grant(b, pa)
             send pa to b in an IPC
  process b:
             put 0x6000 -> pa in private table
             ...

A cool thing you could do w/ exokernel-style memory
  databases like to keep a cache of disk pages in memory
  problem on traditional OS:
    assume an OS with demand-paging to/from disk
    if DB caches some disk data, and OS needs a phys page,
      OS may page-out a DB page holding a cached disk block
    but that's a waste of time: if DB knew, it could release phys
      page w/o writing, and later read it back from DB file (not paging area)
  1. exokernel needs phys mem for some other app
  2. exokernel sends DB a PleaseReleaseAPage() upcall
  3. DB picks a clean page, calls DeallocPage(pa)
  4. OR DB picks dirty page, writes to disk, then DeallocPage(pa)

Exokernel cpu interface
  what does the paper mean by exposing cpu to app?
    kernel upcall to app when it is taking away cpu
    kernel upcall to app when it gives cpu to app
  so if app is running and timer interrupt causes end of slice
    cpu interrupts from app into kernel
    kernel jumps back into app at "please yield" upcall
    app saves state (registers, EIP, &c)
    app calls Yield()
  when kernel decides to resume app

```
    kernel jumps into app at "resume" upcall
    app restores saved registers and EIP
  the only time app registers must be saved is when app calls yield()
    exokernel does not need to save/restore user registers (except PC)
    this makes syscall/trap/contextswitch fast

A cool thing an app can do with exokernel cpu management
  suppose time slice ends in the middle of
    acquire(lock);
    ...
    release(lock);
  you don't want the app to hold the lock despite not running!
    then maybe other apps can't make forward progress
  so the "please yield" upcall can first complete the critical section

Fast RPC with direct cpu management
  how does traditional OS let apps communicate?
    pipes (or sockets)
    picture: two buffers in kernel, lots of copying and system calls
    RPC probably takes 8 kernel/user crossings (read()s and write()s)
  how does exokernel help?
    Yield() can take a target process argument
      almost a direct jump to an instruction in target process
      kernel allows only entries at approved locations in target
    kernel leaves regs alone, so can contain arguments
      (in constrast to traditional restore of target's registers)
    target app uses Yield() to return
    so only 4 crossings
  note RPC execution just appears in the target!
    *not* a return from read() or ipc_recv()

summary of low-level performance ideas
  mostly about fast system calls, traps, and upcalls
    system call speed is, sadly, very important
    slowness encourages complex system calls, discourages frequent calls
  trap path doesn't save most registers
  some sys calls don't use a kernel stack
  fast upcalls to user space (no need for kern to restore regs)
  protected call for IPC (just jump to known address; no pipe or send/recv)
  map some kernel structures into user space (pg tbl, reg save, ...)

bigger ideas -- mostly about abstractions
  it's a win for applications to construct their own big abstractions
    can customize for power and performance
    apps need low-level operations for this to work
  much of kernel can be implemented at user-level
    while preserving sharing and security
    very surprising
  protection does *not* require kernel to implement big abstractions
    e.g. can protect process pages w/o kernel managing address spaces
    1997 paper develops this fully for file systems
  address space abstraction can be decomposed
    into phys page allocation and va->pa mappings
  no need to preserve serial thread of control for a process/thread
    i.e. no need to have program ask for inputs via read() or ipc_recv()
    OK for exception or IPC to jump directly into a process
    OK if context switch isn't transparent

what happened to the exokernel?

first, a word about expectations
  this is a research paper
  the paper has some claims and ideas
    so we want to know if the claims turn out to be justified,
    and if the ideas turn out to be useful
  the main claim is that exposing low-level mechanisms and resources
    will allow applications to do new things,
    or help them get better performance
  the ideas are the specific low-level interfaces for VM, context switch,
```

```
    protected call, interrupt dispatch

lasting influence from the exokernel:
  unix gives much more low-level control than it did in 1995
    very important for a small number of applications
  vmm host/guest interfaces are often very physical
  library operating systems are often used, e.g. in unikernels
  people think a lot about kernel extensibility now, e.g. kernel modules
```

6.828 2014 Lecture 16: Singularity
==

Required reading:
    [Singularity](../readings/hunt07singularity.pdf)
        [Language support for message passing](../readings/singularity-eurosys2006.pdf)

Overview
--

Singularity is a Microsoft Research experimental O/S
  many people, many papers, reasonably high profile
  choice of problems maybe influenced by msft experience w/ windows
  we can speculate about influence on msft products

Stated goals
  increase robustness, security
    particularly w.r.t. extensions
  decrease unexpected interactions
  incorporate modern techniques

High level structure
  microkernel: kernel, processes, IPC
  they claim to have factored services into user processes (page 5)
    NIC, TCP/IP, FS, disk driver (sealing paper)
    kernel: processes, memory, some IPC, nameserver
    UNIX compatibility is not a goal, so avoiding some Mach pitfalls
  on the other hand there are 192 system calls (page 5)

Most radical part of design:
  Only one address space (paging turned off, no use of segments)
    kernel and all processes
  User processes run w/ full h/w privs (CPL=0)

Why is that useful?
  Performance
  Fast process switching: no page table switch
  Fast system calls: CALL not INT
  Fast IPC: no copying
  Direct user program access to h/w, for e.g. device drivers
  Table 1 shows they are a lot faster at microbenchmarks

But their main goal wasn't performance!
  robustness, security, interactions

Is *not* using pagetable protection consistent w/ goal of robustness?
  unreliability comes from *extensions*
    browser plug-ins, loadable kernel modules, &c
  typically loaded into host program's address space
    for speed and convenience
  so VM h/w already not relevant
  can we just do without hardware protection?

How would an extension work in Singularity?
  e.g. device driver, new network protocol, browser plug-in
  Separate process, communicate w/ host process via IPC

What do we think the challenges will be for single address space?
  Prevent evil or buggy programs from writing each other or kernel
  Support kill and exit -- avoid entangling

SIP
-

general SIP philosophy:
  "sealed"
  No modification from outside:
    none of JOS calls that take target envid argument (except start/stop)
    probably no debugger

    only IPC
  No modification from within:
    no JIT, no class loader, no dynamically loaded libraries

SIP rules
  only pointers to your own data
    no pointers to other SIP data or into kernel
    thus no sharing despite shared address space!
    limited exception for IPC messages in exchange heap
  SIP can allocate pages of memory from kernel
    different allocations are not contiguous

Why so crucial that SIPs can't be modified? Can't even modify themselves?
  What are the benefits?
    no code insertion attacks
    probably easier to reason about correctness
    probably easier to optimize, inline
      e.g. delete unused functions
    SIP can be a security principle, own files
  Is it worth the pain?

Why not like Java VM, can share all data?
  SIPs rule out all inter-process interactions
    except explicit via IPC
  SIPs more robust
  SIPs let every process have its own language run-time, GC scheme, &c
    though they are trusted and better not have bugs
    equivalent in sensitivity to kernel code
    so will be much harder for people to cook up their own
  SIPs make it easy for kernel to clean up after kill or exit

How to keep SIPs from reading/writing other SIPs?
  Only read/write memory the kernel has given you
  Have compiler generate code to check every access?
    "does this point to memory the kernel gave us?"
    Would slow code down (esp since mem isn't contig)
    We don't trust compiler

PL-based protection
---

the overall structure:
  1. compile to bytecodes
  2. verify bytecodes during install
  3. compile bytecodes -> machine code during install
  4. run the verified machine code w/ trusted runtime
  Why not compile to machine code?
  Why not JIT at run time?
  Why not verify at compile time?
  Why not verify at run time?

What does bytecode verification buy Singularity?
  Does it verify "only r/w memory kernel gave us"?
  Not exactly, but related:
    Only use reachable pointers [draw diagram]
    Cannot create a new pointer
      only trusted runtime can create pointers
    So if kernel/runtime never supply out-of-SIP pointers
      verified SIP can only use its own memory
  What does the verifier have to check to establish that?
    A. Don't cook up pointers (only use pointers someone gives you)
    B. Don't change mind about type
       Would allow violation of A, e.g. interpret int as pointer
    C. Don't use after free
       Re-use might change type, violate B
       Enforced with GC (and exchange heap linearity)
    D. Don't use uninitialized variables
    D. In general, don't trick the verifier
  Example?

```
      R0 <- new SomeClass;
      jmp L1
      ...
      R0 <- 1000
      jmp L1
      ...
    L1:
      mov (R0) -> R1
    potential problem:
      last mov is OK if via 1st jmp (assuming ptr legitimate)
        reads first element of SomeClass
      not OK if via 2nd jmp
        0x1000 may point into kernel
    verifier tries to deduce type for every register
      by pretending to execute along each code path
      requires that all paths to a reg use result in same type
      check that all reg uses OK for type
      would decide R0 has type int, or type SomeClass *
        either way, verifier would say "no"

Bytecode verification seems to do *more* than Singularity needs
  e.g. cooking up pointers might be OK, as long as within SIP's memory
  verifier may forbid some programs that might have been OK on Singularity
  Benefits of full verification:
    Fast execution, often don't need runtime checks at all
      Though some still needed: array bounds, OO casts, stack expansion
    Type check IPC types
    Need to allow r/w of exchange heap, but it is not SIP's memory
    Stack page allocation
    Do sys calls run on stack in SIP's memory?
      prevent thread X from wrecking thread Y's kernel syscall stack

You could put an interpreter in a SIP to evade ban on self-modifying code
  Would that cause trouble?

What parts are trusted vs untrusted?
  That is:
    All s/w has bugs
    Trusted s/w: if it has bugs, it can crash Singularity or wreck other SIPs
    Untrusted s/w: if it has bugs, can only wreck itself
  Let's consider some ordinary app, not a server.
    compiler. compiler output. verifier. verifier output. GC.

Exchange heap
--

Paper also talks about IPC
  How do SIPs communicate?
  endpoints, channels
  recv endpoint is a queue of messages
  message bodies are in "exchange heap"
  cool: no copy

Exchange heap is shared memory!
  What are the dangers?
  send the wrong type of data
  modify my msg to you while you are using it
  modify a totally unrelated message
  use up all exchange heap memory and don't free

How do they prevent abuse via exchange heap?
  verifier ensures SIP bytecodes keep only one ptr to anything in exchange heap
    never e.g. two
    and that SIP doesn't keep ptr after send()
      single-ptr rule helps here
    verifier knows when last ptr goes away
      via send
      via making another exchange heap obj point to it
      via delete
```

single ptr rule prevents change-after-send
  and also ensures delete when done
delete is explicit, no GC, but it's OK
  since verifier guarantees only a single ptr to each block
runtime maintains owning-SIP entry in each exchg heap block
  updates on send() &c
  used to clean up in exit()

What are channel contracts for?
  Are they just nice to have, or do other parts of Singularity rely on them?
  The type signatures clearly are important.
    bytecode verifier (or something similar) must check them.
  The state machine part guarantees finite queues, no blocking send().
    and also catches protocol implementation errors
    e.g. sending msg when not expected

How does receive work?
  checks endpoints in shared mem, block on condition variable if no msgs
  so send must do a wakeup syscall

How do system calls into the kernel work?
  INT? CALL?
  what stack?
  since same stack, how does GC know?
  can a SIP pass pointers to kernel?

Endpoints function as capabilities
  Can pass them
  Can't talk to other SIPs w/o a channel
  Page 5 says they use channels to restrict access to e.g. files

Does evaluation support their claims?
  Robustness?
  Good model for extensions?
  Performance?
    e.g. real win from single address space, cheap syscall, switch, IPC
    Table 1, but only microbenchmarks
    Figure 5: unsafe code tax
      physical memory -- means paging disabled -- is this Singularity?
      Add 4KB pages -- means turn on paging, but single page table, all CPL=0
      separate domain -- separate page table for one of the SIPs, so switching costs
      ring 3 -- CPL=3 thus INT costs (for just one of the SIPs)
      full microkernel -- pgtable+INT for each of three SIPs

```
6.828 2016 Lecture 17: Scalable Locks

Why this paper?
  Figure 2 in the paper -- disaster! (details later)
  the locks themselves are ruining performance
    rather than letting us harness multi-core to improve performance
  this "non-scalable lock" phenomenon is important
  why it happens is interesting and worth understanding
  the solutions are clever exercises in parallel programming

the problem is interaction of locks w/ multi-core caching
  so let's look at the details

back in the locking lecture, we had a fairly simple model of multiple cores
  cores, shared bus, RAM
  to implement acquire, x86's xchg instruction locked the bus
    provided atomicity for xchg

real computers are much more complex
  bus, RAM quite slow compared to core speed
  per-core cache to compensate
  hit: a few cycles
  RAM: 100s of cycles

how to ensure caches aren't stale?
  core 1 reads+caches x=10, core 2 writes x=11, core 1 reads x=?

answer:
  "cache coherence protocol"
  ensures that each read sees the latest write
    actually more subtle; look up "sequential consistency"

how does cache coherence work?
  many schemes, here's a simple one
  each cache line: state, address, 64 bytes of data
  states: Modified, Shared, Invalid [MSI]
  cores exchange messages as they read and write

messages (much simplified)
  invalidate(addr): delete from your cache
  find(addr): does any core have a copy?
  all msgs are broadcast to all cores

how do the cores coordinate with each other?
  I + local read -> find, S
  I + local write -> find, inval, M
  S + local read -> S
  S + local write -> inval, M
  S + recv inval -> I
  S + recv find  -> nothing, S
  M + recv inval -> I
  M + recv find  -> reply, S

can read w/o bus traffic if already S
can write w/o bus traffic if already M
  "write-back"

compatibility of states between 2 cores:

                  core1
                  M S I
                M - - +
        core2   S - + +
                I + + +

invariant: for each line, at most one core in M
invariant: for each line, either one M or many S, never both

Q: what patterns of use benefit from this coherence scheme?
```

```
   read-only data (every cache can have a copy)
   data written multiple times by one core (M gives exclusive use, cheap writes)

other plans are possible
  e.g. writes update copies rather than invalidating
  but "write-invalidate" seems generally the best

Real hardware uses much more clever schemes
  mesh of links instead of bus; unicast instead of broadcast
    "interconnect"
  distributed directory to track which cores cache each line
    unicast find to directory

Q: why do we need locks if we have cache coherence?
   cache coherence ensures that cores read fresh data
   locks avoid lost updates in read-modify-write cycles
     and prevent anyone from seeing partially updated data structures

people build locks from h/w-supported atomic instructions
  xv6 uses atomic exchange
  other locks use test-and-set, atomic increment, &c
  the __sync_... functions in the handout turn into atomic instructions

how does the hardware implement atomic instructions?
  get the line in M mode
  defer coherence msgs
  do all the steps (e.g. read old value, write new value)
  resume processing msgs

what is performance of locks?
  assume N cores are waiting for the lock
  how long does it take to hand off the lock?
    from previous holder to next holder
  bottleneck is usually the interconnect
    so we'll measure cost in terms of # of msgs

what performance could we hope for?
  if N cores waiting,
  get through them all in O(N) time
  so each critical section and handoff takes O(1) time
    i.e. does not increase with N

test&set spinlock (xv6/jos)
  waiting cores repeatedly execute e.g. atomic exchange
  Q: is that a problem?
  yes!
    we don't care if waiting cores waste their own time
    we do care if waiting cores slow lock holder!
  time for critical section and release:
    holder must wait in line for access to bus
    so holder's mem ops take O(N) time
    so handoff time takes O(N)

Q: is O(N) handoff time a problem?
   yes! we wanted O(1) time
   O(N) per handoff means all N cores takes O(N^2) time, not O(N)

ticket locks (linux):
  goal: read-only spin loop, rather than repeated atomic instruction
  goal: fairness (turns out t-s locks aren't fair)
  idea: assign numbers, wake up one at a time
    avoid constant t-s atomic instructions by waiters
  Q: why is it cheaper than t-s lock?
  Q: why is it fair?
  time analysis:
    what happens in acquire?
      atomic increment -- O(1) broadcast msg
        just once, not repeated
      then read-only spin, no cost until next release
```

what happens after release?
invalidate msg for now_serving
N "find" msgs for each core to read now_serving
so handoff has cost O(N)
note: it was *reading* that was costly!
oops, just as bad O() cost as test-and-set

jargon: test-and-set and ticket locks are "non-scalable" locks
== cost of single handoff increases with N

is the cost of non-scalable locks a serious problem?
after all, programs do lots of other things than locking
maybe locking cost is tiny compared to other stuff

see paper's Figure 2
let's consider Figure 2(c), PFIND -- parallel find
x-axis is # of cores, y-axis is finds completed per second (total throughput)
why does it go up?
why does it level off?
why does it go *down*?
what governs how far up it goes -- i.e. the max throughput?
why does it go down so steeply?

reason for suddenness of collapse
serial section takes 7% on one core (Figure 3, last column)
so w/ 14 cores you'd expect just one or two in crit section
so it seems odd that collapse happens so soon
BUT:
once P(two cores waiting for lock) is substantial,
critical section + handoff starts taking longer
so starts to be more than 7%
so more cores end up waiting
so N grows, and thus handoff time, and thus N...

some perspective
acquire(l)
x++
release(l)
surely a critical section this short cannot affect overall performance?
takes a few dozen cycles if same core last held the lock (still in M)
everything operates out of the cache, very fast
a hundred cycles if lock is not held and some other core previously held it
10,000 if contended by dozens of cores
many kernel operations only take a few 100 cycles total
so a contended lock may increase cost not by a few percent
but by 100x!

how to make locks scale well?
we want just O(1) msgs during a release
how to cause only one core to read/write lock after a release?
how to wake up just one core at a time?

Idea:
what if each core spins on a *different* cache line?
acquire cost?
atomic increment, then read-only spin
release cost?
invalidate next holder's slots[]
only they have to re-load
no other cores involved
so O(1) per release -- victory!
problem: high space cost
N slots per lock
often much more than size of protected object
(this solution is due to Anderson et al.)

MCS
[diagram, code in handout]
goal: as scalable as anderson, but less space used

  idea: linked list of waiters per lock
  idea: one list element per thread, since a thread can wait on only one lock
    so total space is O(locks + threads), not O(locks*threads)
  acquire() pushes caller's element at end of list
    caller then spins on a variable in its own element
  release() wakes up next element, pops its own element
  change in API (need to pass qnode to acquire and release to qnode allocation)

performance of scalable locks?
  figure 10 shows ticket, MCS, and optimized backoff
  # cores on x-axis, total throughput on y-axis
  benchmark acquires and releases, critical section dirties four cache lines
  Q: why doesn't throughput go up as you add more cores?
  ticket is best on two cores -- just one atomic instruction
  ticket scales badly: cost goes up with more cores
  MCS scales well: cost stays the same with more cores

Figure 11 shows uncontended cost
  very fast if no contention!
  ticket:
    acquire uses a single atomic instruction, so 10x more expensive than release
    some what more expensive if so another core had it last

Do scalable locks make the kernel scale better?
  No. Scalability is limited by length of critical section
    Scalable locks avoid collapse
  To fix scalability, need to redesign kernel subsystem
    An example of redesign in next lecture

Notes on Linux and MCS locks (thanks to Srivatsa)

The Linux kernel has scalable (or non-collapsing) locks and is using it. One of the earliest efforts to fix the performance of ticket-based spinlocks dates back to 2013[1], and appears to have used the paper we read today as motivation. (But that particular patchset never actually made it to the mainline kernel). MCS locks found their way into the mutex-lock implementation at around the same time[2].

Replacing ticket-spinlocks with scalable locks, however, turned out to be a much harder problem because locking schemes such as MCS would bloat the size of each spin-lock, which was undesirable due to additional constraints on the spin-lock size in the Linux kernel.

A few years later, the Linux developers came up with 鈥渜spinlocks鈥� (which use MCS underneath, with special tricks to avoid bloating the size of the spin-lock) to replace ticket-spinlocks, which is now the default spin-lock implementation in the Linux kernel since 2015[3][4]. You may also find this article[5] (written by one of the contributors to the Linux locking subsystem) quite interesting in this context.

The old and unused ticket-spinlock implementation was deleted from the codebase in 2016[6].

[1]. Fixing ticket-spinlocks:
      https://lwn.net/Articles/531254/
      https://lwn.net/Articles/530458/

[2]. MCS locks used in the mutex-lock implementation:
      http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=2bd2c92cf07cc4a

[3]. MCS locks and qspinlocks:
      https://lwn.net/Articles/590243/

[4]. qspinlock (using MCS underneath) as the default spin-lock implementation:
      http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=a33fda35e3a765

[5]. Article providing context and motivation for various locking schemes:

http://queue.acm.org/detail.cfm?id=2698990

[6]. Removal of unused ticket-spinlock code from the Linux kernel:
http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=cfd8983f03c7b2

6.828 2016 Lec 18: Scaling OSes
==

Plan
 Scaling OSes
 Concurrent hash tables
 Reference counters
 RadixVM
 Scalability commutativity rule

Scaling kernels
--

Goal: scale with many cores
  Many applications rely heavily on OS
        file system, network stack, etc.
  If OS isn't parallel, then apps won't scale
  ==> Execute systems calls in parallel

OS evolution for scaling
  Early versions of UNIX big kernel lock
  Fine-grained locking and per-core data structures
  Lock-free data structures
  Today: case study of scaling VM subsystem

Problem: sharing
  OS maintains many shared data structures
    proc table
        buffer cache
        scheduler queue
        etc.
  They protected by locks to maintain invariants
  Application may contend on the locks
  Limits scalability

Extreme version: share nothing (e.g., Barrelfish, Fos)
   Run an independent kernel on each core
   Treat the shared-memory machine as a distributed systems
     Maybe even no cache-coherent shared memory
   Downside: no balancing
     If one 1 cores has N threads, and other cores have none
   => Someone must worry about sharing
  Today's focus: use shared-memory wisely

Example: process queue
  One shared queue
  Every time slice, each core invokes scheduler()
  scheduler() locks scheduling queue
  If N is large, invocations of scheduler() may contend
  Contention can result in dramatic performance collapse
    Scalable locks avoid collapse
        But still limits nuber of scheduler() invocations per sec.

Observation: many cases where sharing is unintended
  The threads are *not* sharing
     N cores, N threads
         Can run each thread on its own core
   Can we void sharing when apps are not sharing?

One idea: avoiding sharing in common case
   Each core maintains its own queue
   scheduler() manipulates its own queue
   No sharing -> no contention

Concurrent hash tables
--

Hash tables

Used for shared caches (name cache, buffer cache)
  map block# to block
Implementations
  one lock per hash table (bad scaling)
    lots of unintended sharing
  one lock per bucket (better scaling)
    blocks that map to same buck share unintendedly
  lock-free lists per bucket (see below)
    little unintended sharing
      case: search with on list, while someone is removing

Lock-free bucket

```
    struct element {
  int key;
  int value;
  struct element *next;
};

struct element *bucket;

    void push(struct element *e) {
    again:
        e->next = bucket;
        if (cmpxchg(&bucket, e->next, e) != e->next)
            goto again;
}

struct element *pop(void) {
    again:
        struct element *e = bucket;
        if (cmpxchg(&bucket, e, e->next) != e)
            goto again;
        return e;
 }
```

No changes to search
  More complicated to remove from middle, but can be done

Challenge: Memory reuse (ABA problem)
  stack contains three elements
    top -> A -> B -> C
  CPU 1 about to pop off the top of the stack,
    preempted just before cmpxchg(&top, A, B)
  CPU 2 pops off A, B, frees both of them
    top -> C
  CPU 2 allocates another item (malloc reuses A) and pushes onto stack
    top -> A -> C
  CPU 1: cmpxchg succeeds, stack now looks like
    top -> B -> C
  this is called the "ABA problem"
  (memory switches from A-state to B-state and back to A-state without being able to tell)

Solution: delay freeing until safe
  E.g., Arrange time in epochs
  Free when all processors have left previous

Reference counters
--

Challenge: involves true sharing
  Many resources in kernel are reference counted
  Often a scaling bottleneck (once unintended sharing is removed)

Reference counter
  Design 1: inc, dec+iszero in lock/unlock
    content on cache-line for lock
  Design 2: atomic increment/decrement

```
    content on cache-line for refcnt
  Design 3: per-core counters
    inc/dec: apply op to per-core value
    iszero: add up all per-core values and check for zero
          need per-core locks for each counter
    space overhead is # counters * # cores

Refcache
  An object has a shared reference count
  Per-core cache of deltas
    inc/dec compute a per-core delta
    iszero(): applies all deltas to global reference counter
  If global counter drops to zero, it stays zero
  Space
    Uncontended reference counters will be evicted from cache
    Only cores that use counter have delta

Challenge: determining if counter is zero
  Don't want to call iszero() on-demand
    it must contact all cores
  Idea: compute iszero() periodically
    divide time into epochs (~10 msec)
    at end of an epoch core flushes deltas to global counter
    if global counter drops to zero, put on review queue for 2 epochs later
      if no core has it on its review queue
    2 epochs later: if global counter is still zero, free object
        why wait 2 epochs?
  See example in paper in Figure 1
  More complications to support weak references

Epoch maintainance
  global epoch = min(per-core epochs)
  each core periodically increase per-core epoch
        each 10 msec call flush()+review()
  one core periodically compute global epoch

RadixVM
--

Case study of avoiding unintended sharing
 VM system (ops: map, unmap, page fault)
 Challenges:
   reference counters
   semantics of VM operations
   when unmap returns page must be unmapped at all cores

Goal: no unintended sharing for VM ops
  Ops on different memory regions
  No cache-line transfer when no sharing
  Ok to have sharing when memory regions overlap
    That is intended sharing

mmap has several usages:
  Grow the address space with new memory
  Map files into the address space
  Share memory between processes (e.g., for libraries)

What data structures does VM subsystem need to maintain?
  Table with hardware pages (e.g., ppinfo array in jos)
  Table to map va to pa (e.g., x86 page tables)
  Table with info for each mapped region
     the range of VAs for a region
         a pointer to the inode for the mapped file
  Table to map pa to va  (e.g., for swapping out a physical page)

Modern OSes use an index data structure to present table w. mapped regions
  For example, a balanced tree in Linux and FreeBSD
  Lock-free balanced trees are tricky. Linux has a lock per tree instead.
  Lock-free trees are also not a solution: unintended sharing (see figure 6)
```

Paper's solution: radix tree
  Behaves like hardware page tables
  Disjoint lookups/inserts will access disjoint parts of the tree (see figure 7)
    stores a separate copy of the mapping metadata for each page
    metadata also stores pointers to physical memory page
  Folds repeated entries
  No range queries
  Count the number of uses slots in a radix node using refcache

TLB shootdown
  Unmap requires that no core has the page mapped before returning
  The core running the unmap must send TLB shootdowns to the cores that have page in their TLB
    Some application don't share every memory page
    Nice if we could send shootdowns just to the cores that used the page
  Which cores do have the page in their TLB?
    Easy to determine if the processor has a software-filled TLB
        At a TLB misses the kernel can record the core # for the page
        x86 has a hardware-filled TLB
  Solution: per-core page tables
    The paging hardware will set the accesses bit only in the per-core page table

Maps/unmaps for overlapping region
  simple: locks enforce ordering of concurrent map/unmap on overlapping regions
    acquire locks, going left to right
  page-fault also takes a lock

Implementation
  sv6 (C++ version of xv6)

Eval
  Metis and microbenchmarks

Avoiding unintentional sharing
--

Scalable commutativity rule
  rule: if two operations, commute then there is a scalable (conflict-free) implementation
  non-overlapping map/unmap are example of a general rule
  intuition:
        if ops commute, order doesn't matter
        communication between ops must be unnecessary
  http://pdos.csail.mit.edu/papers/commutativity:sosp13.pdf

Notes
--

Description of the bug mentioned in the change log (from git log):

    Previously, we didn't flush zero deltas. Remarkably, this is wrong
    and we only realized this today. Consider

```
        t ->
core 0     - * +   |   - * +   |   - * +   |
     1   +       - * +       - * +       - *
global  1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
true    1 2 1 1 2 1 1 2 1 1 2 1 1 2 1 1 2 1 1
epoch   ^-----1----^-----2-----^-----3-----^
```

    At the end of epoch 3, the object's global count will have been zero
    and stayed zero for two epochs, even though its true count is not
    zero.

This paper requires understanding
  - Modern VM designs that support mmap/munmap
  - RCU concurrent data structures
Should we add papers the following papers?
  - Mach VM paper as a tutorial of modern VM design (perhaps during xv6 lectures)

- Some RCU paper

6.828 2011 Lecture 19: Virtual Machines

Read: A comparison of software and hardware techniques for x86
virtualizaton, Keith Adams and Ole Agesen, ASPLOS 2006.

what's a virtual machine?
  simulation of a computer
  running as an application on a host computer
  accurate
  isolated
  fast

why use a VM?
  one computer, multiple operating systems (OSX and Windows)
  manage big machines (allocate CPUs/memory at o/s granularity)
  kernel development environment (like qemu)
  better fault isolation: contain break-ins

how accurate do we need?
  handle weird quirks of operating system kernels
  reproduce bugs exactly
  handle malicious software
    cannot let guest break out of virtual machine!
  usual goal:
    impossible for guest to distinguish VM from real computer
    impossible for guest to escape its VM
  some VMs compromise, require guest kernel modifications

VMs are an old idea
  1960s: IBM used VMs to share big machines
  1990s: VMWare re-popularized VMs, for x86 hardware

terminology
  [diagram: h/w, VMM, VMs..]
  VMM ("host")
  guest: kernel, user programs
  VMM might run in a host O/S, e.g. OSX
    or VMM might be stand-alone

VMM responsibilities
  divide memory among guests
  time-share CPU among guests
  simulate per-guest virtual disk, network
    really e.g. slice of real disk

why not simulation?
  VMM interpret each guest instruction
  maintain virtual machine state for each guest
    eflags, %cr3, &c
  much too slow!

idea: execute guest instructions on real CPU when possible
  works fine for most instructions
  e.g. add %eax, %ebx
  how to prevent guest from executing privileged instructions?
    could then wreck the VMM, other guests, &c

idea: run each guest kernel at CPL=3
  ordinary instructions work fine
  privileged instructions will (usually) trap to the VMM
  VMM can apply the privileged operation to *virtual* state
    not to the real hardware
  "trap-and-emulate"

Trap-and-emulate example -- CLI / STI
  VMM maintains virtual IF for guest
  VMM controls hardware IF
    Probably leaves interrupts enabled when guest runs
    Even if a guest uses CLI to disable them

```
  VMM looks at virtual IF to decide when to interrupt guest
  When guest executes CLI or STI:
    Protection violation, since guest at CPL=3
    Hardware traps to VMM
    VMM looks at *virtual* CPL
      If 0, changes *virtual* IF
      If not 0, emulates a protection trap to guest kernel
  VMM must cause guest to see only virtual IF
    and completely hide/protect real IF

trap-and-emulate is hard on an x86
  not all privileged instructions trap at CPL=3
    popf silently ignores changes to interrupt flag
    pushf reveals *real* interrupt flag
  all those traps can be slow
  VMM must see PTE writes, which don't use privileged instructions

what real x86 state do we have to hide (i.e. != virtual state)?
  CPL (low bits of CS) since it is 3, guest expecting 0
  gdt descriptors (DPL 3, not 0)
  gdtr (pointing to shadow gdt)
  idt descriptors (traps go to VMM, not guest kernel)
  idtr
  pagetable (doesn't map to expected physical addresses)
  %cr3 (points to shadow pagetable)
  IF in EFLAGS
  %cr0 &c

how can VMM give guest kernel illusion of dedicated physical memory?
  guest wants to start at PA=0, use all "installed" DRAM
  VMM must support many guests, they can't all really use PA=0
  VMM must protect one guest's memory from other guests
  idea:
    claim DRAM size is smaller than real DRAM
    ensure paging is enabled
    maintain a "shadow" copy of guest's page table
    shadow maps VAs to different PAs than guest
    real %cr3 refers to shadow page table
    virtual %cr3 refers to guest's page table
  example:
    VMM allocates a guest phys mem 0x1000000 to 0x2000000
    VMM gets trap if guest changes %cr3 (since guest kernel at CPL=3)
    VMM copies guest's pagetable to "shadow" pagetable
    VMM adds 0x1000000 to each PA in shadow table
    VMM checks that each PA is < 0x2000000

Why can't VMM just modify the guest's page-table in-place?

also shadow the GDT, IDT
  real IDT refers to VMM's trap entry points
    VMM can forward to guest kernel if needed
    VMM may also fake interrupts from virtual disk
  real GDT allows execution of guest kernel by CPL=3

note we rely on h/w trapping to VMM if guest writes %cr3, gdtr, &c
  do we also need a trap if guest *read*s?

do all instructions that read/write sensitive state cause traps at CPL=3?
  push %cs will show CPL=3, not 0
  sgdt reveals real GDTR
  pushf pushes real IF
    suppose guest turned IF off
    VMM will leave real IF on, just postpone interrupts to guest
  popf ignores IF if CPL=3, no trap
    so VMM won't know if guest kernel wants interrupts
  IRET: no ring change so won't restore restore SS/ESP

how can we cope with non-trapping instructions that reveal real state?
  modify guest code, change them to INT 3, which traps
```

```
  keep track of original instruction, emulate in VMM
  INT 3 is one byte, so doesn't change code size/layout
  this is a simplified version of the paper's Binary Translation

how does rewriter know where instruction boundaries are?
  or whether bytes are code or data?
  can VMM look at symbol table for function entry points?

idea: scan only as executed, since execution reveals instr boundaries
  original start of kernel (making up these instructions):
  entry:
    pushl %ebp
    ...
    popf
    ...
    jnz x
    ...
    jxx y
  x:
    ...
    jxx z
  when VMM first loads guest kernel, rewrite from entry to first jump
    replace bad instrs (popf) with int3
    replace jump with int3
    then start the guest kernel
  on int3 trap to VMM
    look where the jump could go (now we know the boundaries)
    for each branch, xlate until first jump again
    replace int3 w/ original branch
    re-start
  keep track of what we've rewritten, so we don't do it again

indirect calls/jumps?
  same, but can't replace int3 with the original jump
  since we're not sure address will be the same next time
  so must take a trap every time

ret (function return)?
  == indirect jump via ptr on stack
  can't assume that ret PC on stack is from a call
  so must take a trap every time. slow!

what if guest reads or writes its own code?
  can't let guest see int3
  must re-rewrite any code the guest modifies
  can we use page protections to trap and emulate reads/writes?
    no: can't set up PTE for X but no R
  perhaps make CS != DS
    put rewritten code in CS
    put original code in DS
    write-protect original code pages
  on write trap
    emulate write
    re-rewrite if already rewritten
    tricky: must find first instruction boundary in overwritten code

do we need to rewrite guest user-level code?
  technically yes: SGDT, IF
  but probably not in practice
  user code only does INT, which traps to VMM

how to handle pagetable?
  remember VMM keeps shadow pagetable w/ different PAs in PTEs
  scan the whole pagetable on every %cr3 load?
    to create the shadow page table

what if guest writes %cr3 often, during context switches?
  idea: lazy population of shadow page table
  start w/ empty shadow page table (just VMM mappings)
```

  so guest will generate many page faults after it loads %cr3
  VMM page fault handler just copies needed PTE to shadow pagetable
    restarts guest, no guest-visible page fault

what if guest frequently switches among a set of page tables?
  as it context-switches among running processes
  probably doesn't modify them, so re-scan (or lazy faults) wasted
  idea: VMM could cache multiple shadow page tables
    cache indexed by address of guest pagetable
  start with pre-populated page table on guest %cr3 write
  would make context switch much faster

what if guest kernel writes a PTE?
  store instruction is not privileged, no trap
  does VMM need to know about that write?
    yes, if VMM is caching multiple page tables
  idea: VMM can write-protect guest's PTE pages
  trap on PTE write, emulate, also in shadow pagetable

this is the three-way tradeoff the paper talks about
  trace costs / hidden page faults / context switch cost
  reducing one requires more of the others
  and all three are expensive

how to guard guest kernel against writes by guest programs?
  both are at CPL=3
  delete kernel PTEs on IRET, re-install on INT?

how to handle devices?
  trap INB and OUTB
  DMA addresses are physical, VMM must translate and check
  rarely makes sense for guest to use real device
    want to share w/ other guests
    each guest gets a part of the disk
    each guest looks like a distinct Internet host
    each guest gets an X window
  VMM might mimic some standard ethernet or disk controller
    regardless of actual h/w on host computer
  or guest might run special drivers that jump to VMM

Today's paper

Two big issues:
  How to cope with instructions that reveal privileged state?
    e.g. pushf, looking at low bits of %cs
  How to avoid expensive traps?

VMware's answer: binary translation (BT)
  Replace offending instructions with code that does the right thing
    Code must have access to VMM's virtual state for that guest

Example uses of BT
  CLI/STI/pushf/popf -- read/write virtual IF
  Detect memory stores that modify PTEs
    Write-protect pages, trap the first time, and rewrite
    New sequence modifies shadow pagetable as well as real one

How to hide VMM state from guest code?
  Since unprivileged BT code now reads/writes VMM state
  Put VMM state in very high memory
  Use segment limits to prevent guest from using last few pages
  But set up %gs to allow BT code to get at those pages

BT challenges
  Hard to find instruction boundaries, instructions vs data
  Translated code is a different size
    Thus code pointers are different
    Program expects to see original fn ptrs, return PCs on stack
    Translated code must map before use

```
    Thus every RET needs to look up in VMM state

Intel/AMD hardware support for virtual machines
  has made it much easier to implement a VMM w/ reasonable performance
  h/w itself directly maintains per-guest virtual state
    CS (w/ CPL), EFLAGS, idtr, &c
  h/w knows it is in "guest mode"
    instructions directly modify virtual state
    avoids lots of traps to VMM
  h/w basically adds a new priv level
    VMM mode, CPL=0, ..., CPL=3
    guest-mode CPL=0 is not fully privileged
  no traps to VMM on system calls
    h/w handles CPL transition
  what about memory, pagetables?
    h/w supports *two* page tables
    guest page table
    VMM's page table
    guest memory refs go through double lookup
      each phys addr in guest pagetable translated through VMM's pagetable
    thus guest can directly modify its page table w/o VMM having to shadow it
      no need for VMM to write-protect guest pagetables
      no need for VMM to track %cr3 changes
    and VMM can ensure guest uses only its own memory
      only map guest's memory in VMM page table
```

```
6.828 2016 Lecture 19: Virtual Machines, Dune

Read: Dune: Safe User-level Access to Privileged CPU features, Belay et al,
OSDI 2012.

Plan:
  virtual machines
  x86 virtualization and VT-x
  Dune

*** Virtual Machines

what's a virtual machine?
  simulation of a computer, accurate enough to run an O/S

diagram: h/w, host/VMM, guest kernels, guest processes
  VMM might run in a host O/S, e.g. OSX
    or VMM might be stand-alone

why are we talking about virtual machines?
  VMM is a kind of kernel -- schedules, isolates, allocates resources
  modern practice is for VMM and guest O/S to cooperate
  can use VMs to help solve O/S problems

why use a VM?
  run lots of guests per physical machine
    often individual services requires modest resources
    would waste most of a dedicated server
    for cloud and enterprise computing
  better isolation than processes
  one computer, multiple operating systems (OSX and Windows)
  kernel development environment (like qemu)
  tricks: checkpoint, migrate, expand

VMs are an old idea
  1960s: IBM used VMs to share big machines
  1990s: VMWare re-popularized VMs, for x86 hardware

how accurate must a VM be?
  usual goal:
    impossible for guest to distinguish VM from real computer
    impossible for guest to escape its VM
  must allow standard O/S to boot, run
  handle malicious software
    cannot let guest break out of virtual machine!
  some VMs compromise, require guest kernel modifications

why not simulation (e.g, Qemu)?
  VMM interprets each guest instruction
  maintain virtual machine state for each guest
    eflags, %cr3, &c
  correct but slow

idea: execute guest instructions on real CPU
  works fine for most instructions
    e.g. add %eax, %ebx
  what if the guest kernel executes a privileged instruction?
    e.g. IRET into user-level, or load into %cr3
    can't let the guest kernel really run at CPL=0 -- could break out

idea: run each guest kernel at CPL=3
  ordinary instructions work fine
  privileged instructions will (usually) trap to the VMM
  VMM emulates
    maybe apply the privileged operation to the virtual state
    maybe transform e.g. page table and apply to real hardware
  "trap-and-emulate"

what x86 state must a trap-and-emulate VMM "virtualize"?
```

```
  b/c guest can't be allowed to see/change the real machine state.
  %cr3
  pagetable
  IDT
  GDT
  CPL
  IF in EFLAGS
  %cr0 &c

Trap-and-emulate example -- CLI / STI
  VMM maintains virtual IF for guest
  VMM controls hardware IF
    Probably leaves interrupts enabled when guest runs
    Even if a guest uses CLI to disable them
  VMM looks at virtual IF to decide when to interrupt guest
  When guest executes CLI or STI:
    Protection violation, since guest at CPL=3
    Hardware traps to VMM
    VMM looks at *virtual* CPL
      If 0, changes *virtual* IF
      If not 0, emulates a protection trap to guest kernel
  VMM must cause guest to see only virtual IF
    and completely hide/protect real IF

trap-and-emulate is hard on an x86
  not all privileged instructions trap at CPL=3
  popf silently ignores changes to interrupt flag
  pushf reveals *real* interrupt flag
  can solve with binary translation, but complex
  page table is the hardest to virtualize efficiently
    VMM must install a modified copy of guest page table
    VMM must see guest writes to guest PTEs!

*** Hardware-supported virtualization

VT-x/VMX/SVM: hardware supported virtualization
  success of VMs resulted Intel and AMD adding support for virtualization
  makes it easy to implement virtual-machine monitor

VT-x: root and non-root mode
  diagram: VMCS, EPT, %cr3, page table
  VMM runs in VT-x "root mode"
    can modify VT-x control structures such as VMCS and EPT
  Guest runs in non-root mode
    has full access to the hardware, with privilege
      CPL=0, %cr3, IDT, &c
    but VT-x checks some operations and exits to VMM
  New instructions to change between root/non-root mode
    VMLAUNCH/VMRESUME: root -> non-root
    VMCALL: non-root -> root
    plus some interrupts and exceptions cause VM exit
  VM control structure (VMCS)
    Contains state to save or restore during transition
    Configuration (e.g., trap to root mode on page fault, or not)

for our pushf/popf interrupt-enable flag example
  guest uses the hardware flag
    pushf, popf, eflags, &c read/write the flag
      as long as CPL=0
    hardware seems to the guest to act normally
  when a device interrupt occurs
    VMCS lets VMM configure whether each interrupt goes to guest or host
    if guest:
      hardware checks guest interrupt-enable flag
      hardware vectors through guest IDT &c
      no need for exit to VMM
    if host:
      VT-x exits to VMM, VMM handles interrupt
```

```
VT-x: page tables
  EPT -- a *second* layer of address translation
  EPT is controlled by the VMM
  %cr3 is controlled by the guest
  guest virtual -cr3-> guest physical -EPT-> host physical
    %cr3 register holds a guest physical address
    EPT register holds a host physical address
  EPT is not visible to the guest
  so:
    guest can freely read/write %cr3, change PTEs, &c
      hardware sees these changes just as usual
    VMM can still provide isolation via EPT
  typical setup:
    VMM allocates some RAM for guest to use
    VMM maps guest physical addresses 0..size to RAM, in the EPT
    guest uses %cr3 to configure guest process address spaces

what prevents the guest from mapping and accessing the host's memory?
  and thus breaking isolation?

how to handle devices?
  VT-x selectively allows INB and OUTB
  also need to translate DMA addresses
    when guest wants to provide address of a DMA buffer to a device
    VT-d provides a mapping system for DMA devices to use
  but: rarely makes sense for guest to use real device
    want to share w/ other guests
    each guest gets a part of the disk
    each guest looks like a distinct Internet host
    each guest gets an X window
  VMM usually mimics some standard ethernet or disk controller
    regardless of actual h/w on host computer
  or guest might run special drivers that jump to VMM

*** Dune

the big idea:
  use VT-x to support Linux processes, rather than guest O/S kernels
  then process has fast direct access to %cr3, IDT, &c
  might allow new uses of paging not possible with Linux
  these goals are similar to those of the Exokernel

the general scheme -- diagram
  Dune is a "loadable kernel module" for Linux
  an ordinary process can switch into "Dune mode"
  a Dune-mode process is still a process
    has memory, can make Linux system calls, is fully isolated, &c
  but:
    isolated w/ VT-x non-root mode
      rather than with CPL=3 and page table protections
    memory protection via EPT -- Dune only adds
      entries referring to physical pages allocated
      to that process.
    system call via VMCALL (rather than INT)

why is it useful for Dune to use VT-x to isolate a process?
  process can manage its own page table via %cr3
    since it runs at CPL=0
  fast exceptions (page fault) via its own IDT
    no kernel crossings!
  can run sandboxed code at CPL=3
    so process can act like a kernel!

Example: sandboxed execution (paper section 5.1)
  suppose your web browser wants to run a 3rd-party plug-in
    it might be malicious or buggy
  browser needs a "sandbox"
    execute the plug-in, but limit syscalls / memory accesses
  assume browser runs as a Dune process:
```

```
    [diagram: browser CPL=0, plug-in CPL=3]
    create page table with PTE_U mappings for allowed memory
      and non-PTE_U mappings for rest of browser's memory
    set %cr3
    IRET into untrusted code, setting CPL=3
    plug-in can read/write image memory via page table
    plug-in can try to execute system calls
      but they trap into the browser
      and the browser can decide whether to allow each one
  can you do this in Linux?
    these specific techniques are not possible in Linux
      there's no user-level use of CPL, %cr3, or IDT
    fork, set up shared memory, and intercept syscalls
      but it's a pain

Example: garbage collection (GC)
  (modified Boehm mark-and-sweep collector)
  GC is mostly about tracing pointers to find all live data
    set a mark flag in every reached object
    any object not marked is dead, and its memory can be re-used
  GC can be slow b/c tracing pointers can take 100s of milliseconds
  The scheme:
    Mutator runs in parallel with tracer -- with no locks
    At some point the tracer has followed all pointers
      But the mutator may have modified pointers in already-traced objects
      It might have added a pointer that makes some unmarked object actually live
    Pause the mutator (briefly)
    Look at all pages the mutator has modifed since tracer started
      Re-trace all objects on those pages
  How does Dune help?
    Use PTE dirty bit (PTE_D) to detect written pages
    Clear all dirty bits when GC is done
    So program needs quick read and write access to PTEs

As with Exokernel, better user-level access to VM could help many programs
  see e.g. Appel and Li citation

How might Dune hurt performance?
  Table 2
    sys call overhead higher due to VT-x entry/exit
    faults to kernel slower, for same reason
    TLB misses slower b/c of EPT
  But they claim most apps aren't much affected
    b/c they don't spend much time in short syscalls &c
    Figure 3 shows Dune within 5% for most apss in SPEC2000 benchmark
      exceptions take lots of TLB misses

How much can clever use of Dune speed up real apps?
  Table 5 -- sped up web server w/ Wedge by 20%
  Table 6 -- GC
    overall benefit depends on how fast the program allocates
    huge effect on allocation-intensive micro-benchmarks
    no win for the only real application (XML parser)
      doesn't allocate much memory (so no win from faster GC)
      EPT overhead does slow it down
      but many real apps allocate more than this

How might Dune allow new functionality?
  sandboxing via CPL=3 and pagetable
  sthreads -- pagetable per thread, rather than per process
  and speed alone might make some ideas (GC, DSM, &c) feasible

Dune summary
  Dune implements processes with VT-x rather than ordinary page table
  Dune processes can use both Linux system calls AND privileged h/w
  allows fast process access to page tables and page faults
  allows processes to build kernel-like functionality
    e.g. separate page table per thread, or CPL=3 sandboxes
    hard to do this at all (let alone efficiently) with ordinary processes
```

6.828 2016 Lecture 20: O/S Network Performance, IX

Reading: IX: A Protected Dataplane Operating System for
         High Throughput and Low Latency, OSDI 2014

this lecture
  O/S network stack performance
  IX as case study

O/S net stacks are complex, many goals
  lots of protocols: TCP, UDP, IP routing, NFS, ping, ARP
  portability across lots of device drivers
  code tends to be modular and general-purpose ,
  in-kernel to enforce protections e.g. protect port 80
  in-kernel to de-multiplex, e.g. ARP vs TCP

today: focus on design for high performance servers
  e.g. memcached
  high request rate -- single Amazon page has 100s of items
  often short requests
  lots of clients, lots of potential parallelism
  want high request rate under high load
  want low latency under low/modest load
  TCP for reliability

what are the relevant h/w limits?
  i.e. what should we hope for?

throughput limits:
  10 gigabit ethernet: 12.5 million 100-byte packets/second
  40 gigabit ethernet: 50   million 100-byte packets/second
  RAM: a few gigabytes per second
  interrupts: a million per second
  system calls: a few million per second
  contended locks: a million per second
  inter-core data movement: a few million per second
  so:
    if limited by ethernet and RAM, 10 million/sec short queries
    if limited by interrupts, locks, &c: 1 million/sec (maybe per core)

latency limits:
  latency important for e.g. web page with 100s of items
  low load:
    network speed-of-light and switch round-trip time
    interrupt
    queue operations
    sleep/wakeup
    system calls
    inter-core data movement
  high load:
    latency determined by # waiting to be served (queuing)
    usually a throughput issue -- higher efficiency keeps queues shorter
  latency is hard to reason about, hard to improve

what does JOS Lab 6 do?
  [e1000, kernel driver and DMA rings, input and output helpers,
    network server, applications]
  Polls for input; no interrupts; maybe wasteful
  Copies data at least once (kernel -> helper)
  Lots of IPCs
  Lots of context switches
  Lots of enqueue/dequeue
  Little multi-core parallelism
  Clear opportunities for throughput improvement!

what does Linux do?
  [diagram]
  queue: NIC DMA
  processing: driver interrupt

```
  queue: input queue
  processing: TCP (or UDP, ICMP, NFS, &c)
  queue: socket buffer (store until app wants to read)
  processing: application read()

potential Linux problems:
  (most of these have at least partial fixes in modern Linux)
  interrupt per packet is expensive
  system call per message is expensive
  multi-core sharing: queues, TCP connection tables, packet buffer free list
  how to split load of processing incoming packets among cores?
  how to avoid expensive inter-core hand-off of packet data?
  what happens under high input load?
    livelock, early queues grow, not many CPU cycles to drain them

IX: a case study of a (different) high-performance stack
  OSDI 2014
  overlap with Dune authors
  TCP only
  lives in Linux
  different syscall API (doesn't preserve Linux API)
  different stack architecture (doesn't use Linux stack code or design)

IX assumptions
  dedicated server
  multiple server threads
  lots of concurrent clients -> parallelism
  clients are independent -> parallelism
  little processing per request (e.g. no disk read)

IX diagram -- Figure 1(a)
  Linux kernel
  application at CPL=3, multi-threaded
  IX per application, at CPL=0
    IX talks to NIC and NIC DMA queues
  IX in Dune for development convenience
    could equally well be in the kernel

IX's key techniques:
  * batching system call interface
  * process-to-completion
  * polling
  * NIC RSS + no inter-core interaction in stack
  * zero copy

batching syscall interface
  why useful?
    syscall overhead is big if messages are small
  run_io()
    app gives IX a bunch of writes to multiple TCP connections
    returns a bunch of new data from multiple TCP connections
    (really a bit more general, e.g. returns new connection events too)
    so: one syscall does lots of work!
      no others needed for ordinary operation
  libix presents compatible POSIX socket calls
  one run_io() outstanding per server thread

process-to-completion -- Figure 1(b)
  what does it mean?
    complete the processing of one input before starting on next input
    really complete: driver, TCP, application, enqueue reply
  how?
    run_io(), function call down to driver, return pkt all the way up to app
    app calls next run_io() with reply message
  why?
    single thread carries the packet through all steps
    avoids queues, sleep/wakeup, context switch, inter-core xfers
    helps keep active packet in the CPU data cache (vs long queues)
    avoids livelock if input rate is high
```

```
  thus one thread per core; no context switches

polling rather than interrupts
  what is polling?
    periodically check DMA queues for new input or completed output
    versus interrupts
  why good?
    throughput: eliminates expensive per-packet interrupts
    latency: frequent polling has low latency
  why hard?
    where to put the checks? i.e. in what loop?
    might check too often -- waste CPU
    might check too rarely -- high latency
  IX's solution:
    each core dedicated to one application thread
      while(1) { run_io(); ... }
    run_io() polls NIC DMA queues
    no waste: if no input, nothing for the core to do anyway
    if input, grabs a batch and returns it to application
    polls more frequently at low load, less at high load
      very nice; paper calls this "adaptive polling"

what about multi-core parallelism?
  why needed?
    one core often can't deliver enough throughput
      will leave most of 10-gigabit ethernet idle
      cheaper to buy more cores than more servers (up to a point)
    app code can often run in parallel for different clients
    IX TCP can run in parallel for different connections
  what are the dangers?
    moving TCP/IP stack control info among cores, and locking it
    moving packet data among cores
    application may need to lock; nothing IX can do

IX depends on NIC with RSS -- "receive side scaling"
  modern NICs support many independent DMA queues
    IX tells NIC to set up one queue pair per core
  NIC hashes client IP addr / port to pick the queue
    "flow-consistent hashing"
    thus all packets for a given TCP connection handed to same core!
    no need to share TCP connection state among cores
    no need to move packet data between cores
  run_io() looks at NIC DMA queue for just its own core
  a new connection is given to the core determined by the NIC's hash
    hopefully uniform and results in balanced load

zero copy
  how to avoid IX/user and user/IX copies of TCP data?
    across the CPL=0/CPL=3 boundary (like user/kernel)
    40 gigabits/sec may stress RAM throughput
  IX uses page table to map packet buffers into both IX and application
    NIC DMAs to/from this memory
    run_io() carries pointers into this memory
  isolation?
    separate RSS NIC queue per application/core
    separate set of buffers per application/core
    presumably sensitive IX/TCP/NIC meta-data is not mapped
    Linux stack has totally separate NIC queues and buffers
  app/IX cooperate to note when received/sent buffer is free
    via run_io()

Evaluation
  what should we look for?
  high throughput under high load -- especially for small messages
  low latency under light load
  throughput proportional of # of cores
  compatible with real applications

Figure 2
```

- looks at latency under light load
- one client, ping-pong, one request outstanding at a time
- x-axis is message size
- y-axis is throughput (gigabits/second)
- why does the line rise with increasing message size?
  - lots of fixed overheads amortized over increasing data
  - rtt, TCP/IP header, interrupts (for Linux), syscalls, TCP processing
- what limits the rise?
  - 10-gigabit ethernet minus headers
- why does IX beat Linux?
  - for small messages (e.g. 64 bytes):
    - latency-limited
    - IX polling sees the message sooner
    - IX has no interrupt/queueing/sleep/wakeup
    - fewer system calls
    - paper says 5.7 us for IX 64-byte; 24 us for Linux
      - this is enough for people to care!
  - for big messages:
    - throughput-limited
    - IX has less advantage here, since most wins are per-packet
    - IX's zero-copy might be important

Figure 3(a)
- effect of adding cores on throughput
- ideally: throughput proportional to core count
- 18 clients, 64-byte messages, one per connection
  - (is this enough clients to keep many cores busy?)
- x-axis is number of cores
- y-axis is RPCs/second in millions
- why do the lines go up (at least at start)?
  - work is split over more parallel cores
- is the throughput proportional to the number of cores?
  - probably "yes" for all of them, at start
  - so locking &c are not causing problems
- note half a million / second for IX with one core
  - that's 2 microseconds of CPU time per request/response
  - or about 4000 CPU cycles
- why does IX 10 gigabit line level off?
- why does IX beat Linux?
  - polling, batched system calls, process to completion
- it's impressive that IX still scales linearly at 4 million/sec
  - that's a very high number for any system!
  - it suggests that parallelization is nearly perfect
    - RSS helps
    - s/w must have absolutely no locks, no inter-core data movement

could IX's ideas be adopted in Linux?
- a direct port is possible but maybe not very elegant
  - two stacks (could not share TCP/IP code)
  - two different APIs
- the "control plane" part would have to be developed, might be hard
  - if e.g. multiple applications/services running
  - IX needs dedicated cores!
- but some individual ideas could be (or have been) used in Linux
  - polling NIC drivers
  - batched system calls
  - zero-copy in some situations (e.g. sendfile())
  - RSS (but hard to get perfect scaling)
  - (process-to-completion probably too hard)