- goals:
  - Study different approaches for building an OS
  - Understand the tradeoffs between safety and performance
- What problems does an O/S solve?
  - that is, why not directly program the bare hardware?
  - there are a lot of painful h/w details
  - you'd have to build up lots of functionality for yourself
  - it's unlikely that a computer could run more than one app
- What's the O/S solution?
  - e.g. Windows, Linux
  - the small view: a h/w management library
  - the big view: physical machine -> abstract one w/ better properties
  - layer picture:
    - h/w: CPU, mem, disk
    - kernel: [various services]
    - user: applications, e.g. vi and gcc
  - we care a lot about the interfaces and internel kernel structure
- what services does an O/S kernel typically provide?
  - processes
  - memory
  - file contents
  - directories and file names
  - interprocess communication
  - many others: users, security policies, network, time, terminals
- what makes a good kernel service design?
  - Abstract the hardware for programmer convenience
  - Multiplex the hardware among multiple applications
  - Isolate applications to contain bugs
  - Allow sharing among applications

Outline

- PC architecture
- x86 instruction set

PC architecture

- A full PC has:
  - one or more x86 CPUs, each containing:
    - integer registers (can you name them?) and execution unit
    - floating-point/vector registers and execution unit(s)

        - memory management unit (MMU)
        - multiprocessor/multicore: local interrupt controller (APIC)
    - memory
    - disk (IDE, SCSI, USB)
    - 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`
- `}`
- Draw figure with common bus, I/O, and CPU. The CPU has registers, cache, etc.
- Draw figure showing EIP and how it gets incremented automatically after executing each 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 ...
- 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, so 80386 added support for 32-bit data and addresses (1985)
    - boots in 16-bit mode, bootasm.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 bootasm.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`
- ...and even 32 bits eventually wasn't enough, so AMD added support for 64-bit data addresses (1999)
    - registers are 64 bits wide, called RAX, RBX, etc.
    - 8 more general-purpose registers: R8 thru R15
    - boot: *still* go thru 16-bit and 32-bit modes on the way!

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*
- Use example of a function foo() calling a function bar() with two arguments. Assume that bar returns the sum of the two arguments; write bar's code in C and assembly, and explain the conventions.
- Formally, introduce the conventions. 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
- Discuss the frame pointer, and why it is needed --- so that the name of an argument, or a local variable does not change throughout the function body. Discuss the implications of using a frame pointer on the prologue and epilogue of the function body.
- 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
```

    - function epilogue can easily find return EIP on stack:

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

- `popl %ebp`

- The frame pointer (in `ebp`) is not strictly needed because a compiler can compute the address of its return address and function arguments based on its knowledge of the current depth of the stack pointer (in `esp`).
- The frame pointer is useful for debugging purposes, especially the current backtrace (function call chain) can be computed by following the frame pointers. The current function is based on the current value of `eip`. The current value of `*(ebp+4)` provides the return address of the caller. The current value of `*((*ebp) + 4)` (where `*ebp` contains the saved `ebp` of the caller) provides the return address of the caller's caller, the current value of `*(*(*ebp) + 4)` provides the return address of the caller's caller's caller, and so on . . .
- Discuss the code for the "backtrace" function in gdb.
- 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