Operating Systems Engineering
[OSE, 236376, Spring 2016]

Intro, PC hardware, x86, OS design

Lecturer: Dan Tsafrir
Reception: Wed 18:30, Taub 611

TA: Igor Smolyar
Grader: Marina Minkin
Why do the course?

- **Israeli industry invests monumental efforts**
  - In low-level system-software R&D

- **Big industry players:**
  - Intel, IBM, Google, Redhat, Yahoo, Oracle, Cisco,…

- **Smaller:**
  - Many, many more…

- **Israeli academia invests much less**
  - Strange discrepancy… 😞
  - All Israeli universities, no exceptions 😞
  - Industry reluctantly resorts to training employees themselves 😞
Why do the course?

- **Our goal**
  - Doing OSE will be a distinguishing factor for job interviews

- **Interviewer: “know low-level system software?”**
  - You: “Yes! I’ve implemented an OS from scratch!” 😊
  - Increase chances of getting hired 😊
  - Increase chances of higher pay 😊
Why do the course?

Do you want to

❖ Have an interesting professional life & do cool things
❖ Have potential coming up with your own startup
❖ Live for a while in NY, SF, & such like
❖ Travel all over the world for free on a regular basis
❖ Do a cool research-y job at places like MS research / Intel research / IBM research...

?
Why do the course?

- ...Then consider MSc/PhD in computer systems research
  - Succeeded to publish in prestigious computer systems venues?
  - You become irresistible to U.S.-based companies the likes of MS research, Google, VMware, Apple, Intel and such like

- Dan: “are you good enough?”
  - You: “Yes! I’ve implemented an OS from scratch!” 😊

- Did a postdoc in computer systems?
  - Higher chance of getting accepted to Israeli academia
  - Reason: hungry for excellent systems researchers
  - (Still must be excellent though...)
Why do the course?

- **Yes, there’s a price**
  - Learning something hard is... hard
  - There will be (a little) blood, (some) sweat, and (hopefully not many) tears

- **But**
  - You’ll have fun
  - Satisfaction guaranteed

- **Load**
  - We’re aware of our reputation – it’s no longer true
  - We’ve done a lot to reduce the load & make sure its reasonable
  - Speak up if you feel it isn’t: we’re listening, and we care
LET’S BEGIN
Computer systems

Sequoia - BG/Q
(3rd; 1.6M cores; 17PF/s)
LLNL

Titan - Cray XK7
(2nd; 560K cores; 18PF/s)
Oak Ridge National Lab

Tianhe-2 (MilkyWay-2)
(1st; 3.12M cores; 34PF/s)
China

How to make them do something useful?
Outline

- PC architecture
- x86 instruction set
- GCC calling conventions
- PC emulation
- OSes design

Illustrate a few big CS ideas
BIOS

- **BIOS = Basic I/O System**
  - The BIOS software is saved in ROM
  - Built into the PC and is the first SW ran upon PC powered on
  - The fundamental purposes of the BIOS are
    - To initialize and test the system hardware components, and
    - To load an OS from a non-volatile memory device
  - Provides a consistent way for OSes to interact with the keyboard, display, and other I/O devices

- **Jumpers**
  - Password jumper resets BIOS password
  - CMOS jumper resets BIOS to default
Abstract model

- I/O: communicating data to/from devices
- CPU: digital logic for performing computation
- Memory: $N$ words of $B$ bits
The stored program computer

- Memory holds instructions & data
- CPU is an interpreter of instructions
Memory hierarchy (made-up/approx numbers)

- Registers
  - Fastest memory, closest to the CPU; $\leq 1$ cycle
  - (Assuming 1 GHz processor, every cycle take 1 nanosecond)

- L1
  - First level cache; $\sim 1-5$ cycles

- L2
  - Second level cache; $\sim 10-30$ cycles

- L3 / LLC
  - Third level (typical “last-level”) cache; $\sim 30-100$ cycles

- DRAM
  - Main memory; $\sim 100-500$ cycles

- Disk (paging)
  - $\sim 5$ milliseconds (HDD, SSD is faster)
Memory hierarchy

- CPU and memory hierarchy work in resolution of “cache lines”
  - 32 bytes per line for the x86 32bit arch (!)
  - 64 bytes per line for the x86 64bit arch
x86 (32bit) implementation

- EIP is incremented after each instruction
- Instructions have different lengths
  - Must decode before modifying the EIP
- EIP modified by CALL, RET, JMP, conditional JMP
## x86 registers

General-purpose registers (in that they are readable/writable)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>16 bit</th>
<th>32 bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>16</td>
<td>15</td>
<td>8</td>
<td>7</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>AH</td>
<td>AL</td>
<td></td>
<td></td>
<td>AX</td>
<td>EAX</td>
</tr>
<tr>
<td></td>
<td>BH</td>
<td>BL</td>
<td></td>
<td></td>
<td>BX</td>
<td>EBX</td>
</tr>
<tr>
<td></td>
<td>CH</td>
<td>CL</td>
<td></td>
<td></td>
<td>CX</td>
<td>ECX</td>
</tr>
<tr>
<td></td>
<td>DH</td>
<td>DL</td>
<td></td>
<td></td>
<td>DX</td>
<td>EDX</td>
</tr>
<tr>
<td></td>
<td>BP</td>
<td></td>
<td></td>
<td></td>
<td>EBP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SI</td>
<td></td>
<td></td>
<td></td>
<td>ESI</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DI</td>
<td></td>
<td></td>
<td></td>
<td>EDI</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SP</td>
<td></td>
<td></td>
<td></td>
<td>ESP</td>
<td></td>
</tr>
</tbody>
</table>

- 8, 16, and 32 bit versions
- By convention some registers used for special purposes
  - E.g., eax used to return values
- Example: ADD $10, %eax (other ops: SUB, AND, etc.)
EFLAGS register

- Check whether last arithmetic operation
  - overflowed
  - was positive/negative
  - was [not] zero
  - ....
- whether interrupts are enabled
- ...

- Test ops: TEST EAX, 0
- conditional JMP ops: JNZ <address>
# Addressing modes

<table>
<thead>
<tr>
<th>Mode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>register</td>
<td><code>movl %eax, %edx</code></td>
<td><code>edx = eax</code></td>
</tr>
<tr>
<td>immediate</td>
<td><code>$0x123, %edx</code></td>
<td><code>edx = 0x123</code></td>
</tr>
<tr>
<td>direct</td>
<td><code>0x123, %edx</code></td>
<td><code>edx = *(int32_t*)0x123</code></td>
</tr>
<tr>
<td>indirect</td>
<td><code>( %ebx), %edx</code></td>
<td><code>edx = *(int32_t*)%ebx</code></td>
</tr>
<tr>
<td>displaced</td>
<td><code>4( %ebx), %edx</code></td>
<td><code>edx = *(int32_t*)(%ebx +4)</code></td>
</tr>
</tbody>
</table>

- Memory ops: MOVE, PUSH, POP, etc.
- Most ops can take a memory address
GCC/X86 CALLING CONVENTIONS
x86 dictates stack grows “down” (0 is at the bottom)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
</table>
| 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 instruction
- Used to implement procedure call
gcc dictates how stack is used

- **At function entry (just after call)**
  - %esp points at return address
  - %esp+4 points at first argument
  - %esp+8 to 2\textsuperscript{nd} argument, ...
  - [%ebp, %esp] mark [begin, end] of function’s stack frame
  - %ebp points to saved %ebp from previous function (chain to walk call-stack)

- **At function exit (just after ret)**
  - %esp points to arguments pushed by caller
  - callee may have trashed arguments
  - %eax (+ %edx, if return type is 64-bit) contains return value (or trash if ‘void’)
  - %eax, %ecx, %edx may be trashed
  - %ebp, %ebx, %esi, %edi must contain same values
Example

```c
int main(void) {
    return f(7)+1;
}

int f(int x) {
    return g(x);
}

int g(int x) {
    return x+3;
}
```

```assembly
_main:
    pushl %ebp       // prologue
    movl %esp, %ebp
    push $7           // body
    call _f
    addl $1, %eax
    movl %esp, %ebp   // epilogue
    popl %ebp
    ret

_f:
    pushl %ebp       // prologue
    movl %esp, %ebp
    pushl 8(%esp)     // body
    call _g
    movl %esp, %ebp   // epilogue
    popl %ebp
    ret

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

int main(void) {
    return f(7)+1;
}

int f(int x) {
    return g(x);
}

int g(int x) {
    return x+3;
}
Example

```c
int main(void) {
    return f(7)+1;
}

int f(int x) {
    return g(x);
}

int g(int x) {
    return x+3;
}
```
Example

```c
int main(void) {
    return f(7)+1;
}

int f(int x) {
    return g(x);
}

int g(int x) {
    return x+3;
}
```

```assembly
_main:
    pushl %ebp      // prologue
    movl %esp, %ebp
    push $7         // body
    call _f
    addl $1, %eax
    movl %ebp, %esp  // epilogue
    popl %ebp
    ret

_f:
    pushl %ebp      // prologue
    movl %esp, %ebp
    pushl 8(%esp)    // body
    call _g
    movl %ebp, %esp  // epilogue
    popl %ebp
    ret

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

```c
int main(void) {
    return f(7)+1;
}

int f(int x) {
    return g(x);
}

int g(int x) {
    return x+3;
}
```

```
_main:
pushl %ebp        // prologue
movl %esp, %ebp
push $7           // body
call _f
addl $1, %eax
movl %ebp, %esp   // epilogue
popl %ebp
ret

_f:
pushl %ebp        // prologue
movl %esp, %ebp
pushl 8(%esp)     // body
call _g
movl %ebp, %esp   // epilogue
popl %ebp
ret

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

```c
int main(void) {
    return f(7) + 1;
}

int f(int x) {
    return g(x);
}

int g(int x) {
    return x + 3;
}
```

**_main_**:
```
pushl %ebp   // prologue
movl %esp, %ebp
push $7      // body
call _f
addl $1, %eax
movl %ebp, %esp  // epilogue
popl %ebp
ret
```

**_f_**:
```
pushl %ebp   // prologue
movl %esp, %ebp
pushl 8(%esp)   // body
call _g
movl %ebp, %esp  // epilogue
popl %ebp
ret
```

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

int main(void) {
    return f(7)+1;
}

int f(int x) {
    return g(x);
}

int g(int x) {
    return x+3;
}

_main:
pushl %ebp // prologue
movl %esp, %ebp
push $7 // body
call _f
addl $1, %eax
movl %ebp, %esp // epilogue
popl %ebp
ret

_f:
pushl %ebp // prologue
movl %esp, %ebp
pushl 8(%esp) // body
call _g
movl %ebp, %esp // epilogue
popl %ebp
ret

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

int main(void) {
    return f(7) + 1;
}

int f(int x) {
    return g(x);
}

int g(int x) {
    return x + 3;
}
Example

int main(void) {
    return f(7)+1;
}

int f(int x) {
    return g(x);
}

int g(int x) {
    return x+3;
}
Terminology

- **“caller save” registers**
  - %eax, %ecx, %edx
- **“callee save” registers**
  - %ebp, %ebx, %esi, %edi
- **Functions can do anything that doesn't violate this contract**
I/O – INTERACT WITH OUTSIDE WORLD
4 ways of doing I/O in x86

1. PIO
2. MMIO
3. interrupts
   ❖ Manner by which devices asynchronously communicate with CPU
4. DMA (direct memory access)
   ❖ Allows device to read from / write to DRAM without CPU involvement
Programmed I/O (PIO): example

#define DATA_PORT  0x378
#define STATUS_PORT  0x379
#define CONTROL_PORT  0x37A
#define BUSY 0x80
#define STROBE 0x01

void lpt_putchar(int c) {
    /* wait for printer to consume previous byte */
    while( inb(STATUS_PORT) & BUSY )
    {
    /* 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);
}

- Works like memory accesses but sets I/O signal
- Limited number of I/O addressed ("ports")
- Access with special instructions
  • IN, OUT
Memory Mapped I/O (mmio)

- **How it works**
  - Instead of accessing special ports with IN/OUT
  - Map the registers of the device to memory
  - System controller routes “reads” & “writes” to appropriate device
  - Like magic: addressed, but does not *behave*, like memory
    - Reads/writes have “side effects”
    - Read results can change due to external events

- **Pros**
  - Simplifies programming
    - Programmer uses “normal” physical memory addresses
    - No need for special instructions (e.g., C instead of assembly)
  - Gets around limited size of I/O address space

- **What if mmio address range “hides” DRAM parts?**
  - Then these DRAM parts are not accessible
Memory Mapped I/O (mmio)

32-bit memory mapped devices

Unused

Extended Memory

BIOS ROM

16-bit devices, expansion ROMs

VGA Display

Low Memory

\[ \text{\texttt{< - 0xFFFFFFFF}} \ (4GB) \]
\[ \text{\texttt{< - 0x00100000}} \ (1MB) \]
\[ \text{\texttt{< - 0x000F0000}} \ (960KB) \]
\[ \text{\texttt{< - 0x000C0000}} \ (768KB) \]
\[ \text{\texttt{< - 0x000A0000}} \ (640KB) \]
\[ \text{\texttt{< - 0x00000000}} \]

(This memory layout is dictated by x86, namely, all operating systems running on x86 machines are expecting this layout.)
MMIO interactions
TOOL CHAIN & EMULATION
From C to running program

- Compiler, assembler, linker, and loader...
- “Translation unit”, static vs. external linkage
Emulation

- **PC emulator (QEMU, Bochs)**
  - Does exactly what a real PC would do
  - Only implemented in software
  - (“Bochs” is pronounced “box”)

- **The OS you develop (JOS)**
  - Runs like a normal program in a “host” OS
  - On top of the QEMU emulation layer
  - What are the benefits?
Emulation of HW state/memory

- Stores emulated CPU registers in global variables

```c
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 memory in QEMU’s memory

```c
char mem[256*1024*1024];
```
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;
}
#define KB  1024
#define MB  1024*KB
#define LOW_MEM  640*KB
#define EXT_MEM  10*MB

uint8_t low_mem[LOW_MEM];
uint8_t ext_mem[EXT_MEM];
uint8_t bios_rom[64*KB];
Emulation of x86 memory

```c
uint8_t read_byte(uint32_t pa /* pa = physical address */) {
    if( pa < LOW_MEM )
        return low_mem[pa];
    else if( pa >= 960*KB && pa < MB )
        return rom_bios[pa - 960*KB];
    else if (pa >= MB && pa < MB+EXT_MEM)
        return ext_mem[pa - MB];
    else ...
}

void write_byte(uint32_t pa, uint8_t val) {
    if(phys_addr < LOW_MEM)
        low_mem[pa] = val;
    else if( pa >= 960*KB && pa < MB )
        ; /* ignore attempted write to ROM! */
    else if( pa >= MB && pa < MB+EXT_MEM )
        ext_mem[pa - MB] = val;
    else ...
}
```
Emulation of devices

- **Hard disk** = using a file of the host
- **VGA display** = draw in host’s window
- **Keyboard** = host’s keyboard
- **Clock chip** = host’s timing services
- ...

**Details:**
- Simulate I/O devices by detecting accesses to "special" memory and I/O space and emulating the correct behavior
- E.g., reads/writes from/to emulated hard disk are transformed into reads/writes from/to 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
OS DESIGN
Overall OS Design

- **Two major aspects**
  - What are the main components?
  - What should the interfaces look like?

- **To answer that, need to ask**
  - Why have an OS at all?
  - Why not a library to be linked against apps?

- **Key requirement**
  - Support multiple, concurrent activities
  - Which necessitates resource multiplexing & activity isolation

- **Helpful approach – use abstractions**
  - Disk=>filesystem, NIC=>sockets, CPU+memory=>process
  - Do we always need abstractions?
Kernel types

- **Monolithic kernel**
  - “Big program”, contains all services, runs with full privileges
  - Pros: easy interaction between subsystems
  - Cons: complex => bugs => crashes; no isolation
  - Examples: Windows family, much of the Unix family (Linux, BSD family, Solaris, xv6)

- **Microkernel**
  - “Small program”, running most services as daemons in user space
  - Offers same abstractions, but only kernel runs with full privileges
  - Pro: isolation, can recover from service crashes, can reason about interactions—which become explicit—more easily
  - Con: lots of IPC and context switching
  - Examples: L4 family, QNX, Minix family
Kernel types

- **Exokernel**
  - “Very small program”, no forced abstractions, services (can be) implemented as libraries
  - Security is separated from abstractions; kernel and users can run with “full” privileges
  - Pros: simplicity, performance, freedom of design
  - Cons: portability can be a serious issue (interacting with HW)
  - Example: JOS – the OS you’ll implement

- **Hybrids**
  - Dune, ARKIS – all experimental
  - Linux’s vfio
  - Recent HW support now makes Exokernel concept realistic