Operating Systems Engineering
[OSE, 236376, Spring 2014]

Intro, PC hardware, x86

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

TA: Igor Smolyar
Why do the course?

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

- Major industry players:
  - Intel, IBM, Google, Redhat, Yahoo, Oracle, Cisco,...

- Minor:
  - Many, many more...

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

- **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 😊

- **We aspire**
  - Doing OSE will be a distinguishing factor for job interviews
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 come do a MSc/PhD at my lab!
  - Succeeded to publish in prestigious venues in my area?
  - You become *Irresistible* to U.S.-based companies the likes of MS research, Google, VMware, Apple, Intel and such

- Dan: “are you good enough?”
  - You: “Yes! I’ve implemented an OS from scratch!” 😊
  - (But can’t fool me... 😊)

- Did a postdoc?
  - Higher chance of getting accepted to Israeli academia
  - Reason: hungry for excellent systems researchers
  - (But you still must be excellent...
Why do the course?

- **World-class TAs**
  - No kidding... this is the real deal

- **All course staff members depend on your success**
  - Need people to have enough knowledge to do research at our lab

- **Your success is our success**
  - We want you to succeed
Why do the course?

◆ Yes, there’s a price
  ❖ Learning something hard, is hard

◆ But
  ❖ Have fun
  ❖ Satisfaction guaranteed

◆ Load
  ❖ We’re aware of our reputation
  ❖ 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
Computing systems

K computer (3rd; 11PF/s)  
Sequoia - BG/Q (2nd; 16PF/s)  
Titan - Cray XK7 (1st; 18PF/s)

How to make them do something useful?
Outline

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

Illustrate a few big CS ideas
PC board

- Floppy Connector
- Primary IDE Connector
- SATA Cable Connectors
- Front Panel Connector
- Power Connector
- DIMM_1
- DIMM_2
- DIMM_3
- DIMM_4
- Fan Connector
- Processor
- PCIe x8
- Power Connector
- Password Jumper
- CMOS Jumper
- Battery
- PCIe x1
- PCI 32-bit
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; <= 1 cycle
  - (Assuming 1 GHz processor, every cycle take 1 nanosecond)

- **L1**
  - First level cache; ~ 1–5 cycles

- **L2**
  - Second level cache; ~ 10 – 30 cycles

- **L3 / LLC**
  - Third level (typical “last-level”) cache; ~ 30 – 100 cycles

- **DRAM**
  - Main memory; ~ 100 – 500 cycles

- **Disk (paging)**
  - ~ 5 milliseconds (HDD, SSD is faster)
Memory hierarchy

- CPU and memory hierarchy work in resolution of “cache lines”
  - Say, 32 bytes per line
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>31</th>
<th>16</th>
<th>15</th>
<th>8</th>
<th>7</th>
<th>0</th>
<th>16 bit</th>
<th>32 bit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>AX</td>
<td>EAX</td>
</tr>
<tr>
<td>AL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>BX</td>
<td>EBX</td>
</tr>
<tr>
<td>BL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CX</td>
<td>ECX</td>
</tr>
<tr>
<td>CL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DX</td>
<td>EDX</td>
</tr>
<tr>
<td>DL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>EBP</td>
<td></td>
</tr>
<tr>
<td>SI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ESI</td>
<td></td>
</tr>
<tr>
<td>DI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>EDI</td>
<td></td>
</tr>
<tr>
<td>SP</td>
<td></td>
<td></td>
<td></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>operation</th>
<th>result</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, POS, etc
- Most ops can take a memory address
GCC/X86 CALLING CONVENTIONS
### x86 dictates stack grows down

<table>
<thead>
<tr>
<th>x86 Instruction</th>
<th>Assembly Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>pushl %eax</td>
<td>subl $4, %esp     movl %eax, (%esp)</td>
</tr>
<tr>
<td>popl %eax</td>
<td>movl (%esp), %eax  addl $4, %esp</td>
</tr>
<tr>
<td>call 0x12345</td>
<td>pushl %eip (<em>)    movl $0x12345, %eip (</em>)</td>
</tr>
<tr>
<td>ret</td>
<td>popl %eip (*)</td>
</tr>
</tbody>
</table>

- (*) = 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^{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;
}
```

---

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

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

_g:
  pushl %ebp        // prologue
  movl %esp, %ebp   // prologue
  pushl %ebx        // save %ebx
  movl 8(%ebp), %ebx // body
  addl $3, %ebx     // epilogue
  movl %ebx, %eax   // epilogue
  popl %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
    movl %esp, %ebp
    push $7
    call _f
    addl $1, %eax
    movl %ebp, %esp
    popl %ebp
    ret

_f:
    pushl %ebp
    movl %esp, %ebp
    pushl 8(%esp)
    call _g
    movl %ebp, %esp
    popl %ebp
    ret

_g:
    pushl %ebp
    movl %esp, %ebp
    pushl %ebp
    movl 8(%ebp), %ebx
    addl $3, %ebx
    movl %ebx, %eax
    popl %ebx
    movl %ebp, %esp
    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;
}
```

```c
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
    addl $3, %ebx  // body
    movl %ebx, %eax
    popl %ebx      // restore %ebx
    movl %ebp, %esp // epilogue
    poppl %ebp     // epilogue
    ret
```


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
    pushl $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

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

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

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

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

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

_g:
```assembly
    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;
}
```

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

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

_g:
    pushl %ebp              // prologue
    movl %esp, %ebp        // save %ebx
    pushl %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;
}
```

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

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

_g:
    pushl %ebp
    movl %esp, %ebp
    pushl %ebx
    // save %ebx
    movl 8(%ebp), %ebx
    movl %ebx, %eax
    movl %ebx, %esp
    // restore %ebx
    popl %ebx
    movl %ebp, %esp
    // epilogue
    popl %ebp
    ret
```
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
Programmed I/O (PIO): example

```c
#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

- 0xFFFFFFF (4GB)
- 0x00100000 (1MB)
- 0x000F0000 (960KB)
- 0x000C0000 (768KB)
- 0x000A0000 (640KB)
- 0x00000000

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

- Compiler, assembler, linker, and loader...
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];
```
Emulation of CPU

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;

}
Emulation of x86 memory

#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
END OF LECTURE
Leftover

- **EFLAGS**
  - Every arith op updates EFLAGS
  - Can use bit (e.g., JNZ)

- **Addressing modes**
  - Recall intel assembly (a.k. "NASM") syntax vs. gcc’s
  - Movl = “l” is for long (32bit)
    - in NASM it’d deduced form type of arg, or writing “BYTE PTR” “WORD PTR” before arg
  - Order of args reversed
    - Gnu makes more sense: move ‘src’ to ‘dst’
  - Displaced for structure, cause leftmost arg must be hardcoded
  - Last bullet: as opposed to RISC
Leftover: more memory

- S* can be manipulated
- push $3 actually means (in pseudo C):
  - sp--; *((ss<<4) + sp) = 3;
  - $4096 = 2^{12}; 65536 = 2^{16}$
- If you put sp in ax, then it relates to ds, but typically actually mean ss;
  - to make it work we need either ds=ss, or
  - Use explicit command (? If permitted)
- In protected mode, whereby you have 32bit, we assign ss = ds = cs = ... = 0, to make life easier
- Far pointer is given by both 16bit address + segment, looks like so (in intel notation): 123:456
  - Long jump op requires far pointer
And more memory

- `.code32` doesn’t translate to an assembly command, rather, it tells the assembler to translate the text to binary
  - This is called “assembler directive”
X86 instruction set

- **test:** do “-” rather than “-=“, but update flags; that is, it’s exactly like sub, but doesn’t put result in “dst”
- **Shl = shift left**
- **Rep <op>: loop**