

Section 1: OS Concepts and x86

CS 162

June 24, 2020

Contents

1	Vocabulary	2
2	C Review	3
2.1	Pointer and C Programming Practice	3
3	Fundamental Operating System Concepts	3
4	x86 Assembly	5
4.1	Registers	5
4.2	Syntax	5
4.3	Practice: Clearing a Register	6
4.4	Calling Convention	6
4.5	Instructions Supporting the Calling Convention	6
4.6	Practice: Reading Disassembly	7
4.7	Practice: x86 Calling Convention	8

1 Vocabulary

With credit to the Anderson & Dahlin textbook (A&D):

- **stack** - The stack is the memory set aside as scratch space for a thread of execution. When a function is called, a block is reserved on the top of the stack for local variables and some book-keeping data. When that function returns, the block becomes unused and can be used the next time a function is called. The stack is always reserved in a LIFO (last in first out) order; the most recently reserved block is always the next block to be freed.
- **heap** - **The heap is memory set aside for dynamic allocation.** Unlike the stack, there is no enforced pattern to the allocation and deallocation of blocks from the heap; you can allocate a block at any time and free it at any time.
- **process** - A process is an instance of a computer program that is being executed, typically with restricted rights. It consists of an address space and one or more threads of control. It is the main abstraction for protection provided by the operating system kernel.
- **address space** - The address space for a process is the set of memory addresses that it can use and the state associated with them. The memory corresponding to each process' address space is private and cannot be accessed by other processes, unless it is explicitly shared.
- **C** - A high-level programming language. In order to run it, C will be compiled to low level machine instructions like x86_64 or RISC-V. Note that it is often times easier to express high level ideas in C, but C cannot be used to express many details (such as register allocation).
- **x86** - A very popular family of instruction sets (which includes i386 and x86_64). **Unlike MIPS or RISC-V, x86 is primarily based on CISC (Complex Instruction Set Computing) architecture.** Virtually all servers, desktops, and most laptops (with Intel or AMD) natively execute x86.

2 C Review

2.1 Pointer and C Programming Practice

Write a function that places source inside of dest, starting at the offset position of dest. This is effectively swapping the tail-end of dest with the string contained in source (including the null terminator). Assume both are null-terminated and the programmer will never overflow dest. As an exercise in using pointers, implement it **without using libraries**.

```
void replace(char *dest, char *source, int offset)
{

}

}
```

3 Fundamental Operating System Concepts

1. What are the 3 roles the OS plays?

2. How is a process different from a thread?

3. What is the process address space and address translation? Why are they important?

4. What is dual mode operation and what are the three forms of control transfer from user to kernel mode?

5. Why does a thread in kernel mode have a separate kernel stack? What can happen if the kernel stack was in the user address space?

6. How does the syscall handler protect the kernel from corrupt or malicious user code?

4 x86 Assembly

In the projects for this class, you will write an operating system for a 32-bit x86 machine. The class VM (and probably your laptop) use a 64-bit x86 processor (i.e., an x86-64 processor) that is capable of executing 32-bit x86 instructions. There are significant differences between the 64-bit and 32-bit versions of x86. For this worksheet, **we will focus on the 32-bit x86 ISA** because that is the ISA you will have to read when working on the projects. Remember that if you compile programs on your local machine or directly in the class VM (not in Pintos), the result will be in x86-64 assembly.

4.1 Registers

The 32-bit x86 ISA has 8 main registers: `eax`, `ebx`, `ecx`, `edx`, `esi`, `edi`, `esp`, and `ebp`. You can omit the “e” to reference the bottom half of each register. For example, `ax` refers to the bottom half of `eax`. `esp` is the stack pointer and `ebp` is the base pointer. Additionally, `eip` is the instruction pointer, similar to the program counter in MIPS or RISC-V.

x86 also has *segment registers* (`cs`, `ds`, `es`, `fs`, `gs`, and `ss`) and *control registers* (e.g., `cr0`). You can think of segment registers as offsets when accessing memory in certain ways (e.g., `cs` is for instruction fetches, `ss` is for stack memory), and control registers as configuring what features of the processor are enabled (e.g., protected mode, floating point unit, cache, paging). **We won’t focus on them in this worksheet, but you should know that they exist.** In particular, Pintos sets these up carefully upon startup in `pintos/src/threads/start.S`, so look there if you are interested. Keep in mind that there are special restrictions as to how these registers are used as operands to instructions.

4.2 Syntax

Although the x86 ISA specifies the registers and instructions, there are two different syntaxes for writing them out: Intel and AT&T. Instruction operands are written in a different order in each syntax, which can make it confusing to read one syntax if you are used to the other. For this worksheet, **we will focus on the AT&T syntax** because it is the version used by the toolchain we are using (`gcc`, `as`).

In the AT&T syntax:

- Registers are preceded by a percent sign (e.g., `%eax` for the register `eax`)
- Immediates are preceded by a dollar sign (e.g., `$4` for the constant 4)
- For many (but not all!) instructions, use parentheses to dereference memory addresses (e.g., `(%eax)` reads from the memory address in `eax`)
- You can add a constant offset by prefixing the parentheses (e.g., `8(%eax)` reads from the memory address `eax + 8`)
- Source operands typically precede destination operands, for instructions with two operands.

Instructions are often suffixed by a letter to specify the size of operands. Use the suffix `b` to work with 8-bit *bytes*. Use the suffix `w` to work with 16-bit *words*. Use the suffix `l` to work with 32-bit *longwords* (or *doublewords*). (Analogously, on the x86-64 ISA, append `q` to work with 64-bit *quadwords*). If you omit the suffix, the assembler will add it for you.

Some examples:

- `addw %ax, %bx`: Add the word in `ax` to the word in `bx`, and store the result in `bx`.
- `addl %eax, %ebx`: Add the longword in `eax` to the longword in `ebx`, and store the result in `ebx`.
- `addl (%eax), %ebx`: Add the longword in memory at the address in `eax` to the longword in `ebx`, and store the result in `ebx`.
- `addl 12(%eax), %ebx`: Add the longword in memory at the address `eax + 12` to the longword in `ebx`, and store the result in `ebx`.
- `subl $12, %esp`: Subtract the constant 12 from the longword in `esp`, and store the result in `esp`.

Notice that you don't need special instructions to load from/store to memory. Some other useful instructions are **and**, **or**, and **xor**. An especially common instruction is **mov**:

- `movl %eax, %ebx`: Copy the longword in `eax` into `ebx`.
- `movl $4, %ecx`: Set the longword in `ecx` to 4.
- `movl 4, %ecx`: Read the longword in memory at address 4 and store the result in `ecx`.
- `movl %edx, -8(%ecx)`: Write the longword in `edx` to memory at the address `ecx - 8`.

For the instructions `leal` and `leal`, which you will find in Pintos, the parenthesis notation for memory works differently. They calculate an absolute memory address given a register and offset.

- `leal 8(%eax), %ebx`: Sets `ebx` to `eax + 8`. You can think of this as setting `ebx` to the memory address that `movl 8(%eax), %ebx` would read from.

4.3 Practice: Clearing a Register

Write an instruction that clears register `eax` (i.e., stores zero in `eax`).

4.4 Calling Convention

The **caller** does the following:

1. Push the arguments onto the stack, in reverse order. After this step, the top of the stack must be 16-byte aligned — add padding before pushing arguments, if necessary, so that this is true.
2. Push the return address and jump to the function you are trying to call.
3. When the callee returns, the return address is gone but the arguments are still on the stack.

The **callee** does the following, and must preserve `ebx`, `esi`, `edi`, and `ebp`:

1. (Typical, but not required) Push `ebp` onto the stack, and store current `esp` into `ebp`.
2. Compute the return value and store it in `eax`.
3. Restore `esp` to its value at the time the callee began executing.
4. Pop the return address off of the stack and jump to it.

4.5 Instructions Supporting the Calling Convention

- `pushl %eax` is equivalent to:

```
subl $4, %esp
movl %eax, (%esp)
```

- `popl %eax` is equivalent to:

```
movl (%esp), %eax
addl $4, %esp
```

- `call $0x1234`: push the return address (address of the next instruction of the caller) onto the stack and jump to the specified address (address of the callee).
- `leave` is equivalent to:

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

- **ret** pops a longword off of the stack (typically a return address) and jumps to it.

pushal pushes **eax**, **ecx**, **edx**, **ebx**, **esp**, **ebp**, **esi**, and **edi** to the stack, and **popal** pops values off of the stack and stores them in those registers. They are useful to switch context or handle interrupts.

4.6 Practice: Reading Disassembly

file.c:

```
int global = 0;

int callee(int x, int y) {
    int local = x + y;
    return local + 1;
}

void caller(void) {
    global = callee(3, 4);
}
```

When gcc compiles this file, with optimizations off, it outputs:

file.s:

```
callee:
    pushl    %ebp
    movl     %esp, %ebp
    subl     $16, %esp
    movl     8(%ebp), %edx
    movl     12(%ebp), %eax
    addl     %edx, %eax
    movl     %eax, -4(%ebp)
    movl     -4(%ebp), %eax
    addl     $1, %eax
    leave
    ret

caller:
    pushl    %ebp
    movl     %esp, %ebp
    pushl    $4
    pushl    $3
    call     callee
    addl     $8, %esp
    movl     %eax, global
    nop
    leave
    ret
```

What does each instruction do? Mark the prologue(s), epilogue(s), and call sequence(s).

4.7 Practice: x86 Calling Convention

Sketch the stack frame of `helper` before it returns.

```
void helper(char* str, int len) {  
    char word[len];  
    strncpy(word, str, len);  
    printf("%s", word);  
    return;  
}  
  
int main(int argc, char *argv[]) {  
    char* str = "Hello World!";  
    helper(str, 13);  
}
```