

Section 1: x86, C, and OS Concepts

September 4-6, 2019

Contents

1	Vocabulary	2
2	Warmup	2
2.1	Pointer and C Programming Practice	2
3	x86 Assembly	3
3.1	Registers	3
3.2	Syntax	3
3.3	Practice: Clearing a Register	4
3.4	Calling Convention	4
3.5	Instructions Supporting the Calling Convention	4
3.6	Practice: Reading Disassembly	5
3.7	Practice: x86 Calling Convention	6
4	C Programs	6
4.1	Calling a Function in Another File	6
4.2	Including a Header File	7
4.3	Using <code>#define</code>	7
4.4	Using <code>#include</code> Guards	9
5	Fundamental Operating System Concepts	9

1 Vocabulary

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

- **thread** - A thread is a single execution sequence that can be managed independently by the operating system. (See A&D, 4.2)
- **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.
- **protection** - Protection refers to isolating applications from one another so that a potentially misbehaving application cannot corrupt other applications or the operating system.
- **dual-mode operation** - Dual-mode operation refers to hardware support for multiple privilege levels: a privileged level (called *supervisor-mode* or *kernel-mode*) that provides unrestricted access to the hardware, and a restricted level (called *user-mode*) that executes code with restricted rights.
- **address space** - The address space for a process is the set of memory addresses that it can use. The memory corresponding to each process' address space is private and cannot be accessed by other processes unless it is shared.
- **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's 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.

2 Warmup

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 x86 Assembly

In the projects for this class, you will write an operating system for a 32-bit x86 machine. The class VM (and, likely, your laptop) use a 64-bit x86 processor (i.e., an x86-64 processor) that's 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'll focus on the 32-bit x86 ISA** because that's the ISA you'll 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 for Pintos), the result will be in x86-64 assembly.

3.1 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 they exist.** In particular, Pintos sets these up carefully upon startup in `pintos/src/threads/start.S`, so look there if you're interested. Keep in mind that there are special restrictions as to how these registers are used as operands to instructions.

3.2 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're used to the other. For this worksheet, **we'll focus on the AT&T syntax** because it's 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 (not all!) instructions, use parentheses to dereference memory address (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`.

The instruction `lea`, which you will find in Pintos, is special in that the parenthesis notation for memory works differently. It calculates 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.

3.3 Practice: Clearing a Register

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

3.4 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're 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.

3.5 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` pushes the return address (address of the next instruction of the caller) onto the stack and jumps 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.

3.6 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
```

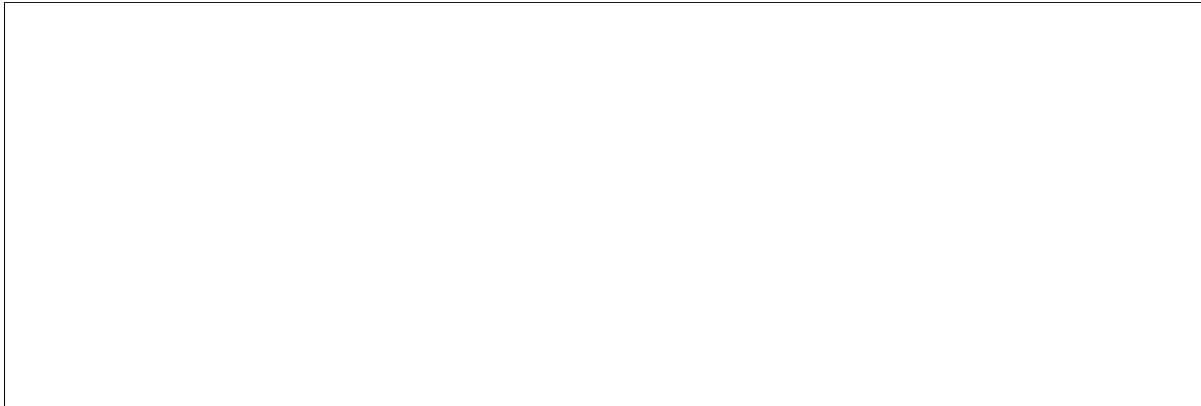
What does each instruction do? Mark the prologue(s), epilogue(s), and call sequence(s).

3.7 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);
}
```



4 C Programs

4.1 Calling a Function in Another File

Consider a C program consisting of two files:

`my_app.c`:

```
#include <stdio.h>

int main(int argc, char** argv) {
    char* result = my_helper_function(argv[0]);
    printf("%s\n", result);
    return 0;
}
```

`my_lib.c`:

```
char* my_helper_function(char* string) {
    int i;
    for (i = 0; string[i] != '\0'; i++) {
        if (string[i] == '/') {
            return &string[i + 1];
        }
    }
    return string;
}
```

```

    }
  }
  return string;
}

```

You build the program with `gcc my_app.c my_lib.c -o my_app`.

1. What is the bug in the above program? (Hint: it's in `my_app.c`.)
2. How can we fix the bug?

4.2 Including a Header File

Suppose we add a header file to the above program and revise `my_app.c` to `#include` it.

`my_app.c`:

```

#include <stdio.h>
#include "my_lib.h"

int main(int argc, char** argv) {
  char* result = my_helper_function(argv[0]);
  printf("%s\n", result);
  return 0;
}

```

`my_lib.h`:

```

char* my_helper_function(char* string);

```

You build the program with `gcc my_app.c my_lib.c -o my_app`.

1. Suppose that we made a mistake in `my_lib.h`, and declared the function as `char* my_helper_function(void);`. Additionally, the author of `my_app.c` sees the header file and invokes the function as `my_helper_function()`. Would the program still compile? What would happen when the function is called?
2. What could the author of `my_lib.c` do to make such a mistake less likely?

4.3 Using #define

Suppose we add a `struct` and `#ifdef` to the header file:

`my_app.c`:

```

#include <stdio.h>
#include "my_lib.h"

int main(int argc, char** argv) {
  helper_args_t helper_args;
  helper_args.string = argv[0];
  helper_args.target = '/';

  char* result = my_helper_function(&helper_args);
  printf("%s\n", result);
}

```

```

    return 0;
}

```

my_lib.h:

```

typedef struct helper_args {
#ifdef ABC
    char* aux;
#endif
    char* string;
    char target;
} helper_args_t;

char* my_helper_function(helper_args_t* args);

```

my_lib.c:

```

#include "my_lib.h"

char* my_helper_function(helper_args_t* args) {
    int i;
    for (i = 0; args->string[i] != '\0'; i++) {
        if (args->string[i] == args->target) {
            return &args->string[i + 1];
        }
    }
    return args->string;
}

```

You build the program with:

```

$ gcc -c my_app.c -o my_app.o
$ gcc -c my_lib.c -o my_lib.o
$ gcc my_app.o my_lib.o -o my_app

```

Convince yourself that this program outputs the same thing as the one in 4.2.

1. What is the size of the `helper_args_t` structure?
2. Suppose we add the line `#define ABC` at the top of `my_lib.h`. Now what is the size of the `helper_args_t` structure?
3. Suppose we leave `my_lib.h` unchanged (no `#define ABC`). But, suppose we instead use the following commands to build the program:

```

$ gcc -DABC -c my_app.c -o my_app.o
$ gcc -c my_lib.c -o my_lib.o
$ gcc my_app.o my_lib.o -o my_app

```

The program will now either segfault or print something incorrect. What went wrong?

4.4 Using #include Guards

Suppose we split `my_lib.h` into two files: `my_helper_function.h`:

```
#include "my_helper_args.h"

char* my_helper_function(helper_args_t* args);
```

`my_helper_args.h`:

```
typedef struct helper_args {
    char* string;
    char target;
} helper_args_t;
```

1. What happens if we include the following two lines at the top of `my_app.c`?

```
#include "my_helper_function.h"
#include "my_helper_args.h"
```

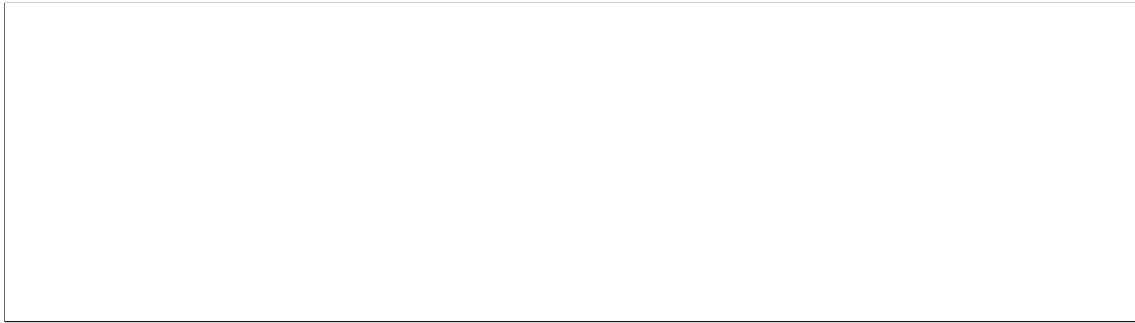
2. How can we fix this? (Hint: look up `#include` guards.)

5 Fundamental Operating System Concepts

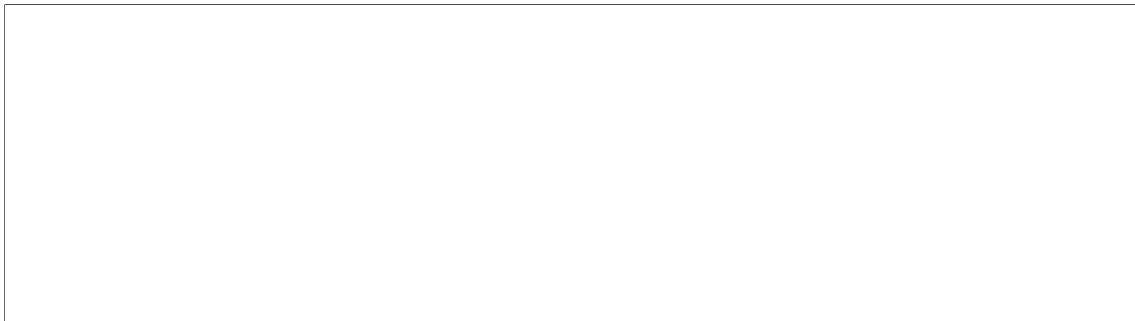
1. What are the 3 roles the OS plays?

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?

