

Lecture 2: Hello, OS World!

Viewing Operating Systems from Multiple Angles

Xin Liu

Florida State University

xliu15@fsu.edu

COP 4610 Operating Systems

<https://xinliulab.github.io/FSU-COP4610-Operating-Systems/>

Programs Running on the Operating System

What is an Operating System?

Operating System: A body of software, in fact, that is responsible for making it easy to run programs (even allowing you to seemingly run many at the same time), allowing programs to share memory, enabling programs to interact with devices, and other fun stuff like that. (Operating Systems: Three Easy Pieces)

To understand an “Operating System”, you must understand what a “program” is

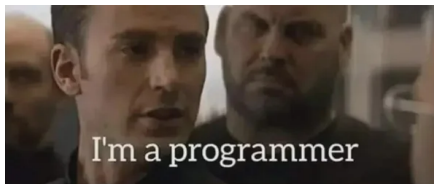
- This course explains OS from an application-driven perspective

Hello, World!

```
#include <stdio.h>
int main()
{
    printf("Hello, World!");
    return 0;
}
```

From this point on,
you began your journey as a
PROGRAMMER.

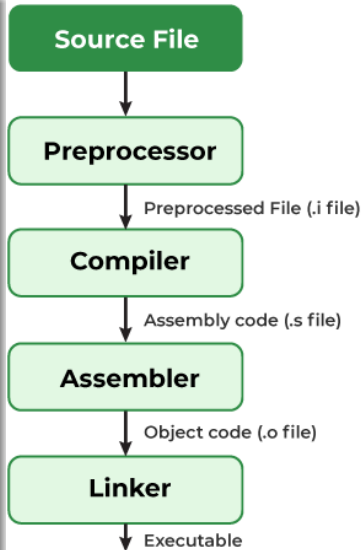
- Try It: [hello.c](#)
- How to make your code readable?
 - Ctrl+Shift+P
 - Search "Format Document".



Each Step of Compilation

What it does and what you get

- 1 **Preprocessor:** Expands macros and includes headers.
`gcc -E hello.c -o hello.i`
Output: `hello.i`
- 2 **Compiler:** Translates C into assembly.
`gcc -S -O0 hello.c -o hello.s`
Output: `hello.s`
- 3 **Assembler:** Converts assembly to an object file.
`gcc -c hello.s -o hello.o`
Output: `hello.o`
- 4 **Linker:** Links object files and libraries into an executable.
`gcc hello.o -o hello`
Output: `hello`



Computer Programs

```
#include <stdio.h>

int main() {
    int a = 1;
    int b = 1;
    int c = a + b;

    printf("%d + %d = %d\n", a, b, c);
}
```

- Try It: [abc.c](#)

Computer: A machine that executes instructions without emotion

- The machine is always "correct"
- If the compiler does not optimize, blue "it executes exactly what we write"

If the compiler does not optimize, “it executes exactly what we write”

- What might a compiler optimize in this code?
- Try It Again: [abc.c](#)

How many compilers are there in our computers?

Understanding “Computer Programs”

Everything is a state machine.

- Every program runs on a computer
- The computer is a state machine
- Program execution is state transition

State Machine Model in C

- **PicoC**: a very small C interpreter for scripting; supports step-by-step execution

```
while (1) {  
    stmt = next_statement();  
    execute(stmt);  
}
```

Can we really turn those imaginative ideas into reality?

- In the AI era: *If you can imagine it, you can build it*
- GDB debugging used to be tedious
- But now, tedious tasks no longer require human labor
 - Even in the future, almost nothing may need humans

Given a Python script that executes GDB step by step and generates a `plot.md`, embedding the state transition of `main()` execution (tracked by local variables only), where each `step` denotes a transition. Executed statements are visualized line by line with highlighted blocks.

Rewrite Any Program Into Non-Recursive Form

Any C Code Can Be Rewritten as Equivalent “SimpleC”

- Each statement does one operation (A function call also counts as one operation)
- Conditional statements contain no operations.
- There is a real tool for this([C Intermediate Language](#)) and an [interpreter](#).

Everything (a C Program) Is a State Machine

- State = variable values + stack
- Initial state = the first statement of `main`
- State transition = execute a small step of one statement

Program = State Machine

A “state machine” is a mathematically rigorous object. This means you can **formally define** it and **reason about it rigorously**.

State:

- A list of stack frames [`StackFrame`, `StackFrame`, ...] plus global variables

Initial State:

- Only one stack frame: `main(argc, argv, PC=0)`
- All global variables are initialized

State Transition:

- Execute the simple statement at `frames[-1].PC`

This Is All of C (Formal Semantics)

With this semantics, we can implement any pure computation:

- From simple to complex: `strlen`, `strstr`, `memcpy`, `sprintf`, ...

But some things *cannot* be implemented:

- Some behaviors of the standard library go beyond “pure computation”
- Examples: `putchar`, `exit`
 - Observation: Pure computation only changes [internal program state](#)
 - But these APIs involve “external state” beyond the program
 - **This is the topic of the Operating Systems course.**

The Smallest Program on an Operating System

Let's try: What Exactly Changes a Program's State from the Outside?

What Is a Program? (After Compilation)

A Minimal CPU State (i.e., your a.out):

```
struct CPUTState {  
    uint32_t regs[32], csrs[CSR_COUNT];  
    uint8_t *mem;  
    uint32_t mem_offset, mem_size;  
};
```

Processor: A cold, instruction-executing [state machine](#)

- Fetch an instruction from $M[PC]$
- Execute it
- Repeat

Can We Gain Control of the Program from the Start?

Building the “Smallest” Program

- Gain control from the very beginning of the program
- According to *Computer Systems: A Programmer's Perspective*, a program starts executing from `_start`
- Try It: [smallest.c](#)

```
void _start() {  
    // ...  
}
```

Let AI Help You Seize Control

- I defined `_start`, how do I compile and run the program directly from `_start`?
 - `gcc -nostartfiles -static -nostdlib smallest.c`

Building the “Smallest” Program

```
void _start() {  
    // ...  
}
```

- But this program causes a Segmentation Fault — why?
- Again, please ask our AI teacher.

```
void _start() {  
    __asm__("mov $60, %eax\n" // syscall: exit  
           "xor %edi, %edi\n" // status: 0  
           "syscall");  
}
```

(Binary) Program = State Machine

State

- Your C code compiles to assembly that executes on a machine state.
- A program state is the combination of memory and registers.
- In `gdb`, you can inspect this state (memory and registers).

Initial State

- Defined by the ABI (e.g., a valid `%rsp`)
- What else is defined by ABI ([Application Binary Interface](#))?

State Transitions

- Executing an instruction
- All instructions change only the program's internal state (memory and registers), except the instruction `syscall`.
 - `gdb` can observe the execution of the state machine step by step
- `syscall` instruction: hands over the state machine to the operating system.

Read the Man Pages

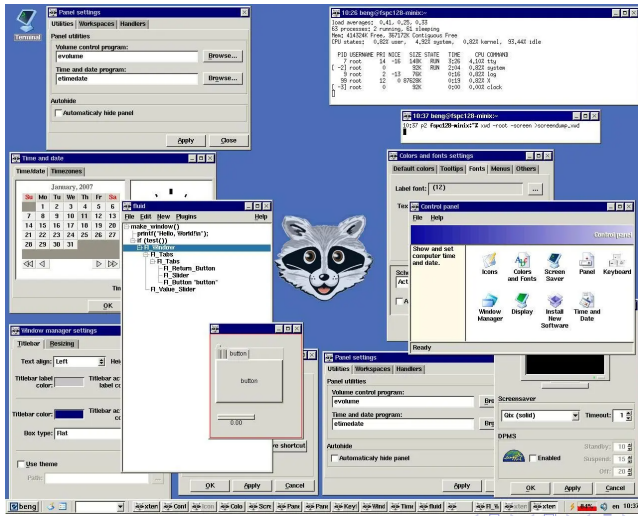
- `man 2 syscalls`: lists all system calls. Online: [man7: syscalls\(2\)](#).
- `man 2 syscall`: describes the generic `syscall()` interface. Online: [man7: syscall\(2\)](#).
 - You will see `x86-64 syscall rax`. This means that on x86/64 the system call number must be placed in `RAX` (64-bit OS) or `EAX` (32-bit OS).
- You can also use [Linux syscall table](#) to see that 60 corresponds to `exit`.
- That is why `"mov $60, %eax\n"`

```
void __start() {  
    __asm__("mov $60, %eax\n" // syscall: exit  
           "xor %edi, %edi\n" // status: 0  
           "syscall");  
}
```

Applications Running on Top of the Operating System

What We Perceive as the “Operating System”

- As users, **we do not perceive the operating system itself.**
- We only interact with **programs running on top of the OS** (processes).



Visible Programs: Applications

Development

- Integrated Development Environments: VSCode, Cursor, ...
- Programming Tools: gcc, clang, nodejs, gdb, ...
- Terminal Tools: tmux, vim, htop, ...

Daily Use

- Office: LibreOffice, GIMP, ...
- Browsers: Chrome, Firefox, ...
- Media: OBS, VLC, ...

Core Utilities (coreutils)

- *Standard* programs for text and file manipulation
- The default installation is [GNU Coreutils](#)
- Lightweight alternatives: [busybox](#), [toybox](#)

System Utilities: Essential and Powerful

- Shell, [binutils](#), ...
- Package management: apt, dpkg, ...
- Networking: ip, ssh, curl, ...
- Multimedia: ffmpeg, gstreamer, ...

Daemon Processes

- The omnipresent `systemd`
 - `systemd-network`, `systemd-logind`, `systemd-udev`, ...
- System management: `cron`, `udisksd`, `unattended-upgrade` (loathed), ...
- Services: `httpd`, `sshd`, ...
- Security modules: `auditd`, `firewalld`, ...
- User services: `pulseaudio`, `dbus-daemon`, ...

Graphics and Media

- Wayland compositor: `xfce4`, `lxde`, ...
- `Pulseaudio`, `pipewire`, `video4linux`, ...

Therefore, all these programs...

Any difference from `4_smallest.c`?

- Short answer: No
- Any program = `4_smallest.c` = state machine

Executable files are OS objects

- Essentially no difference from the binary file `a.out`
- Let's examine the "programs" mentioned above in the command line
 - Don't worry even if it's your first time touching the command line...
 - Ask like a ChatGPT: I have an `a.out` file, how can I explore what's inside?

Takeaways: Everything Is a State Machine

- Both high-level code and machine code can be viewed as state machines.
- A compiler acts as a translator between two types of state machines.
- Without an operating system, state machines can only perform pure computation.
- They can't even communicate results to the outside world.
- The only bridge between a program and the OS is through **system calls**.
- On `x86-64`, this bridge is built on the `syscall` instruction.
- Because system calls are so important, the OS provides tools to observe them.
 - For example, `strace` can track a program's system call sequence during execution.
 - We'll dive into this in the next lecture.