Lecture 12: Bringing it All Together — From Circuits to Execution

(Compilation, Linking, Loading, GOT, and PLT)

Xin Liu

Florida State University
xl24j@fsu.edu

COP 4610 Operating Systems
https://xinliulab.github.io/cop4610.html
September 17, 2025

Hardware: Digital Logic Circuits

- The foundation of the digital system world is built on a single axiom: **NAND gates**.
- Using NAND gates, we can:
  - Construct all other logic gates.
  - Build flip-flops to form memory elements.
  - Create any digital logic circuit, including complex systems like mobile applications.

From Simple to Complex:

- Begin with simple digital circuits.
- Gradually scale up to more abstract systems
  - logic gates $\rightarrow$ circuits $\rightarrow$ machine code $\rightarrow$ high-level programming languages.

A	B	Q
0	0	1
0	1	1
1	0	1
1	1	0

Programming as a State Machine

Key Insight:

- All programs (e.g., C, Java, Python) running on a Von Neumann machine are inherently **state machines**.
- The program state consists of:
  - Local variables, global variables, stack, heap, and the program counter (PC).
- Function calls and returns manipulate the state machine:
  - Call: Pushes a new stack frame.
  - Return: Removes the top stack frame.

Debugging Perspective:

- Understanding programs as state machines provides a clear path for debugging:
  - Identify incorrect state transitions (e.g., unexpected values or infinite loops).
  - Diagnose bugs systematically using the state machine model.

Bridging Programming and Hardware

How High-Level Code Maps to Hardware:

- Programs like `printf("Hello, World!")` are written as text files (byte sequences in ASCII encoding).
- These byte sequences are:
  - Compiled into machine code instructions.
  - Interpreted and executed by circuits formed with NAND gates.
  - High-level code $\rightarrow$ Assembly $\rightarrow$ Machine instructions $\rightarrow$ Logic gates.

Programming vs. Digital Logic:

- Programming introduces abstraction to interact with the underlying digital logic.
- Tools like compilers and operating systems bridge this gap efficiently.

Simple C → Assembly

Instruction Set Architecture

- "Low-level semantics"
- Instructions are based on state machines implemented with logic gates.
- Reference: The RISC-V Instruction Set Manual
  - Easy to implement in hardware.
  - Supports programmable execution effectively.

Compiler (It's a Program Too)

- Translates "high-level state machines"(programs) into "low-level state machines"(instruction sequences).
- Translation principle: Ensures external observable behavior equivalence.

Operating System = Objects + APIs

Application Layer → Libraries → System Calls → Operating System → Device Drivers → Hardware Abstraction Layer → Physical Hardware → Logic Gates

All problems in computer science can be solved by another level of indirection, except for the problem of too many layers of indirection. (David Wheeler)

Implementation: Everything is a State Machine

- A state machine operates according to predetermined logic.
- The key lies in defining and initializing the initial state.

Overview

Executable Files:

- Define the initial state of a state machine.
- Serve as a data structure for process execution.

Key Questions Addressed:

- Q1: How does the operating system load executable files?
- Q2: What is dynamic linking/loading, and how does it work?

Core Topics for This Session

Session Objectives:

- Understand the ELF loader for static binaries.
- Explore dynamic linking and loading mechanisms.

Excutable Linkable File (elf)

Making the Program Recognizable to the Machine

RTFM: Read The ”Fine” Manual

Key Manuals for This Lesson:

- **System V ABI:** Defines the System V Application Binary Interface for the AMD64 architecture, providing essential specifications for binary compatibility.
  - System V ABI (AMD64 Architecture Processor Supplement)
- **Refspecs:** Additional reference specifications to deepen understanding of Linux-based systems.
  - Linux Refspecs

Executable Files: Describing the State Machine

The Operating System: An Execution Environment for Programs (State Machines)

- **Executable File (State Machine Description):**
  - The executable file is a key OS object, describing the initial state and transitions of the program's state machine.
- **Registers:**
  - Most registers are set according to the ABI (Application Binary Interface), with initial setup handled by the OS.
  - For example, the OS initializes the program counter (PC) to start execution.
- **Address Space:**
  - Defined by the binary file and ABI specifications.
  - Includes initial data like `argv` and `envp` (environment variables), along with other necessary information.
- **Additional Information:**
  - The OS may store extra data to aid in debugging and for core dumps in case of errors.

Example: Executable Files on an Operating System

Requirements for an Executable File:

- Must have execution ('x') permission.
- Must be in a format that the loader can recognize as executable.

Example Commands and Output:

```
$ ./a.c
bash: ./a.c: Permission denied

$ ./a.c
bash: ./a.c: Permission denied

$ chmod -x a.out && ./a.out
bash: The file './a.out' is not executable by this user

$ chmod +x a.c && ./a.c
Failed to execute process './a.c'. Reason:
exec: Exec format error
The file './a.c' is marked as an executable but could not be run by
       the operating system.
```

Who Decides If a File is Executable?

The Operating System (OS Code - `execve`) Determines Executability:

- The OS, through `execve`, decides whether a file can be executed.

Try It Out:

- Use `strace` to trace `execve` calls and observe execution failures.
  - `strace ./a.c`
    - Without execute permission on `a.c`: `execve` returns `-1, EACCES`
    - With execute permission but incorrect format on `a.c`: `execve` returns `-1, ENOEXEC`

She-bang (#!/path/to/interpreter):

- The She-bang (`#!`) allows specifying an interpreter for a script or executable.
- She-bang effectively performs a “parameter swap” in `execve`, launching the specified interpreter to execute the file.

Example: Running Python Code in a C File

- Save the Following Code as `helloworld.c`:

```
#! /usr/bin/python3
print("Hello␣World!")
```

- Give the file execute permission:

```
$ chmod +x helloworld.c
```

- Now, you can directly run the `helloworld.c` file to execute the Python code:

```
$ ./helloworld.c
Hello World!
```

Analyzing Executable Files

Binutils - Binary Utilities

GNU Binutils: Essential Tools for Executable Files

- **Creating Executable Files:**
  - `ld` (Linker): Combines object files into a single executable.
  - `as` (Assembler): Translates assembly code into machine code.
  - `ar` and `ranlib`: Manage static libraries.
- **Analyzing Executable Files:**
  - `objcopy`, `objdump`, `readelf`: Inspect and modify executables, often used in computer systems basics.
  - `addr2line`: Maps addresses to line numbers for debugging.
  - `size`, `nm`: Display size information and symbol tables.

Learn More: GNU Binutils Official Page

Why Can We See All This Information?

Debugging Information Added During Compilation:

- When we compile with debug flags, the compiler includes extra information in the binary.
- This information allows tools like `objdump` and `addr2line` to map assembly code back to the original source code.

Example Command:

- Using `gcc -g -S hello.c` generates assembly code with debugging information.
- This enables us to see additional sections in the assembly output, including variable names, line numbers, and other metadata.

Standard of Debugging Information

Mapping Machine State to "C World" State:

- The DWARF Debugging Standard (dwarfstd.org) defines an instruction set, `DW_OP_XXX`, that is Turing Complete.
- This instruction set can perform "arbitrary computations" to map the current machine state back to the C language state.

Challenges and Limitations:

- **Limited Support for Modern Languages:** Advanced features (e.g., C++ templates) are not fully supported.
- **Complexity of Programming Languages:** As languages evolve, it becomes increasingly challenging to accurately map machine states to source code.
- **Compiler Limitations:** Compilers may not always produce perfect debug information, leading to issues like:
  - Frustrating instances of variables being `<optimized out>`
  - Incorrect or incomplete debugging information

Compilation and Linking

Compilation Process: Assembly and Binary

From High-Level Code to Machine Instructions:

- The compiler generates assembly code (text format).
- The assembler converts it into binary instructions.

Example of Generated Assembly and Binary Code:

```
0000000000000000 <main>:
0: f3 0f 1e fa endbr64
4: 48 83 ec 08 sub $0x8,%rsp
8: 31 c0 xor %eax,%eax
a: e8 00 00 00 00 callq ????????
f: 31 c0 xor %eax,%eax
11: 48 83 c4 08 add $0x8,%rsp
15: c3 retq
```

When Address Translation is Unavailable:

- For example, during calls to functions like `hello`.
- Temporary placeholders (e.g., `0x0`) are used until the linker resolves the actual address.

Relocation Logic:

- The 4-byte offset must be filled to satisfy the assertion:

```
assert(
    (char *)hello ==
       (char *)main + 0xf + // call hello's next PC
       *(int32_t *)((uintptr_t)main + 0xb) // offset in call
);
```

Requirements:

- This offset must be written to the executable file.

Relocation Entry in ELF Files

Example Relocation Entry:

```
      Offset            Type          Symbol + Addend
000000000000000b R_X86_64_PLT32    hello - 4
```

Key Points:

- Relocate a 32-bit value to `S + A - P`, where:
  - `S`: Address of the symbol (e.g., `hello`).
  - `A`: Addend specified in the relocation entry.
  - `P`: Address of the relocation itself (`main + 0xb`).
- Understand the behavior of `call S + A - P`.

Understanding Compilation

Compiler (gcc):

- Translates **high-level semantics** (C state machine) into **low-level semantics** (assembly code).

Assembler (as):

- Converts low-level semantics into **binary semantics** (state machine containers).
- Performs a 1-to-1 translation into binary machine code:
  - Includes sections, symbols, and debug information.
- Leaves unresolved information for later (e.g., relocations).

Linker (ld):

- Combines all containers to produce a **complete state machine**.
- Uses:
  - `ldscript` (`-Wl, --verbose`)
  - Links with C Runtime Objects (CRT).
- Resolves errors related to:
  - Missing symbols.
  - Duplicate symbols.

Static ELF Loader

Implementing ELF Loader on an Operating System

Executable Files

- A data structure describing the initial state (transition) of a state machine.
  - Different from in-memory data structures; ”pointers” are replaced by ”offsets.”
  - Defined in parts within `/usr/include/elf.h`.

Loader

- Parses the data structure, copies it into memory, and jumps to the entry point.
- Creates the initial runtime state of the process (`argv`, `envp`, ...).
  - Example: `loader-static.c`
    - Can load any statically linked code such as `minimal-hello.S`.
    - Correctly processes arguments/environment variables.
- **Reference:** System V ABI Figure 3.9 (Initial Process Stack).

Boot Block Loader

Loading the Operating System Kernel?

- It is also an ELF file.
- The process includes:
  - Parsing the data structure.
  - Copying it into memory.
  - Jumping to the entry point.

Example Code: `bootmain.c` (Compatible with `i386/x86-64` architectures)

Linux Kernel: The Grand Entrance

`bootmain.c` and Linux: Any Fundamental Difference? None!

- **Steps to Build the Kernel:**
  - Decompress the source package.
  - Run `make menuconfig` to generate the `.config` file.
  - Compile with `make bzImage -j8`.
  - Optionally, patch the kernel (e.g., `kernel/exit.c`).
- **Compilation Results:**
  - `vmlinux`: ELF-formatted kernel binary.
  - `vmlinuz`: Compressed image for direct QEMU loading.
  - Use `readelf` to view the entry address at `0x1000000` (physical memory 16MB position).
  - `__startup_64`: RTFSC! Start debugging.
- Always remind yourself: **"Don't panic; it's just a state machine"** (just like your lab work!).

Executable Files

Executable File:

- Describes the initial state of a state machine (**program execution**).
- Unlike in-memory data structures:
  - ”Pointers” are replaced by **offsets**.
- Data structure definitions available at:
  - `/usr/include/elf.h`

Loader Implementation

Loader:

- Parses the data structure, copies it into memory, and performs **jumps**.
- Initializes the process runtime environment:
  - `argv`, `envp`, ...

Reference: System V ABI Figure 3.9 (Initial Process Stack).

Dynamic Linking and Loading

The Need for ”Decoupling Applications”

With increasing library functions, the goal is to enable ”runtime linking.”

- **Reducing Redundancy:**
  - Avoid duplication of library functions in every executable file.
  - Ensures efficient disk and memory usage.
- **Versioning Requirements:**
  - Follow fundamental conventions, especially respecting function versions.
  - Employ ”Semantic Versioning” to manage compatibility.
  - Without version control, any new version requires recompiling the **entire program.**
- **Large Project Modularization:**
  - Compile parts of a project without relinking the entire program.
  - Example: Shared libraries like `libjvm.so`, `libart.so`.
  - Analogy: In NEMU, ”plugging the CPU into the board.”

Dynamic Linking

Challenge of Explaining ELF

- Every year: Hard to explain ELF clearly as it becomes overwhelming.
- Root issue: Closely related concepts are forcibly ”dispersed” in data structures.
  - Example: GOT[0], GOT[1], ...

A Simpler Approach

- If the compiler, linker, and loader are under your control:
  - How would you design the most ”intuitive” dynamic linking format?
  - Reflect on improvements to reach ELF!
- Assume the compiler generates **position-independent code (PIC)** for you.

Designing a New Binary File Format

Key Idea: Simplify Dynamic Linking with Symbol Tables

- Streamline dynamic linking by focusing on symbol lookup.

Example Structure:

- **DL_HEAD**
  - `LOAD("libc.dl")` # Load dynamic library
  - `IMPORT(putchar)` # Import external symbols
  - `EXPORT(hello)` # Export symbols for dynamic library
- **DL_CODE**
  - Define the function `hello`:
    - `call DSYM(putchar)` # Dynamic linking to symbol
- **DL_END**

Creating a Toolchain for .dl Files with Minimal Effort

Compiler:

- Start with existing tools:
  - Use `GCC`, `GNU as`.

Binutils:

- Leverage existing tools for basic functionality:
  - `ld = objcopy` (borrowed functionality).
  - `as = GNU as` (also borrowed).
- Additional tools to customize:
  - `readdl` (like `readelf`).
  - `objdump`.
  - You can adapt tools like `addr2line`, `nm`, `objcopy`, etc.

Critical Component: The Loader

- The loader is essential and needs to be built manually.

Dynamic Linking: Implementation

Header Files:

- `dl.h`: Defines the data structures used for dynamic linking.

Toolchain ("All-in-One" Toolkit):

- `dlbox.c`:
  - Includes tools such as `gcc`, `readdl`, `objdump`, and `interp`.

Example Code:

- `libc.S`: Provides implementations for `putchar` and `exit`.
- `libhello.S`: Calls `putchar` and provides `hello`.
- `main.S`: Calls `hello` and provides `main`.
- (Assumes that your high-level language compiler can generate assembly code in this format.)

Reture to ELF

Addressing Design Flaws in .dl Files

Memory Protection and Load Address:

- Allow mapping parts of the .dl file to specific memory locations with controlled permissions (via program header table).

Custom Loader Specification:

- Permit the use of custom loaders instead of relying solely on `dlbox`.
- Introduce `INTERP`.

Space Efficiency:

- Store strings in a constant pool and access them through a unified ”pointer” mechanism.
- This contributes to the complexity of reading ELF files.

Additional Improvements:

- Less critical but still valuable enhancements.
- Follow documentation: `RTFM`/`RTFSC`.

Another Significant Limitation

DSYM as Indirect Memory Access

```
#define DSYM(sym) *sym(%rip)

extern void foo();
foo();
```

One Syntax, Two Scenarios:

- **From Other Compilation Units (Static Linking):**
  - Direct PC-relative jump.
- **Dynamic Linking Libraries:**
  - Symbol resolution is mandatory (cannot be determined at compile-time).

The ”Invention” of GOT & PLT

Our ”Symbol Table” is the Global Offset Table (GOT)

- Now you won’t misunderstand the concept of GOT!
- The concept and name aren’t important; the process of invention is.

Unifying Static and Dynamic Linking: Both Use Static!

- Add a layer of indirection: Procedure Linkage Table (PLT).
- All unresolved symbols are unified and translated into `call`.
- Modern processors optimize this kind of jump (e.g., Branch Target Buffer (BTB)).

Example:

```
putchar@PLT:
   call DSYM(putchar) # in ELF: jmp *GOT[n]

main:
   call putchar@PLT
```

Revisiting `printf`

- You will find that our ”minimal” binary file format is almost exactly the same!
- ELF even includes some additional hacks (e.g., lazy binding).

```
00000000000010c0 <printf@plt>:
 10c0: endbr64
 10c4: bnd jmpq *0x2efd(%rip) # DSYM(printf)
 10cb: nopl 0x0(%rax,%rax,1)

00000000000011c9 <main>:
 ...
 1246: callq 10c0 <printf@plt>
```

One Last Question: Data

- What if we want to reference data in dynamically linked libraries?
  - Data cannot add another layer of indirection
- Issues with `stdout`/`errno`/`environ`:
  - Multiple libraries use them, but there should only be **one copy**!

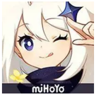
Of course, we did an experiment!

- Use `readelf` to check if `stdout` differs
- Set a watch point in `gdb`
  - It turns out to be treated specially
  - It's a kind of "workaround"

Summary: What is an Executable File?

Before Learning Operating Systems:

- "Something you double-click to open a window."



After Learning Operating Systems:

- An **object** in the operating system (a file).
- A **sequence of bytes** (which can be interpreted as text or encoded symbols).
- A data structure describing the **initial state of a state machine**.

Questions Answered in This Lecture:

- **Q1:** How are executable files loaded by the operating system?
- **Q2:** What are dynamic linking and dynamic loading?

Take-away Messages:

- **Loader:**
  - Uses the underlying mechanism to ”transport” data structures according to specifications
- **Dynamic Linking/Loading:**
  - GOT, PLT, and the smallest binary format