Lecture 15: Program Execution in Memory

(Shell, C Standard Library, Executable and Linkable Format)

Xin Liu

Florida State University

xl24j@fsu.edu

COP 4610 Operating Systems

https://xinliulab.github.io/cop4610.html

October 29, 2024

Outline

Today's Key Question:

How do we make a program run in memory?

Main Topics for Today:

- Shell
  - Starting the Program
- C Standard Library
  - Program Dependencies
- Executable and Linkable Format
  - Making the Program Recognizable to the Machine
- Linking and Loading (To Be Covered After File Management)
  - Combining and Loading Programs into Memory

Shell

Starting the Program

Revisiting the Shell

Shell as an Interface

The Shell provides an interactive interface between the user and the operating system.

How the Shell Executes Programs:

- The user interacts with the shell to start programs.
- When a command is issued by the user in the shell:
  - **Fork:** The OS duplicates a process, creating a child process that is an exact copy of the parent.
  - **Execve:** In the child process, the 'execve' system call replaces the process's memory space with the new program, resetting its state to execute the specified program.
  - **Independent Execution:** The program now runs as a separate process with its own memory space.

Example: Minimal Hello World Program

Do you remember the smallest "Hello World" program we created?

- It has the advantage of being minimal, which allows us to easily observe how a program runs in the operating system.
- Link of Code: **Minimal Hello World Code**

To observe system calls:

- Open two terminals and run the following commands separately:

```
$ gcc minimal_hello.s -c
$ ld minimal_hello.o
$ strace -f -o ./strace.log /bin/sh
$ ./a.out
```

```
$ tail -f ./strace.log
```

Who is the Parent Process?

To identify the parent process of a given PID, run the following command:

- `ps -p <pid> -o pid,ppid,cmd`

Luke, I Am Your Father!

The True Identity of the Shell

How the Shell Executes Programs:

- The user interacts with the shell to start programs.
- When a command is issued, the shell initiates the following process:
  - **Fork:** The shell duplicates the current process, creating a child process that is an exact copy of the parent.
  - **Execve:** In the child process, the execve system call replaces the process's memory space with the new program, resetting its state to execute the specified program.
  - **Independent Execution:** The program now runs as a separate process with its own memory space, while the shell (parent process) waits or handles other tasks.

May The Shell Be With You!

C Standard Library (libc)

Program Dependencies

Why Do We Need `libc`?

- "Bare-metal" Programming: Works (and technically enough), but not user-friendly
- Essential definitions that all programs use:
  - `stddef.h` – Provides types like `size_t`
  - `stdint.h` – Defines standard integer types like `int32_t`, `uint64_t`

```c
#include <stdio.h>
#include <assert.h>

int main() {
      int a;

   printf("Size␣of␣int␣=␣%ld\n", sizeof(int));
   printf("Size␣of␣long␣=␣%ld\n", sizeof(long));

      assert(sizeof(a) == 4);
}
```

Can you guarantee that this code will pass on all machines?

- **Preferred way**: Use fixed-width types, e.g., `int32_t a;` for consistency across platforms.
- **Best practice**: Refer to the official C++ reference for detailed information on types and usage.

Why Do We Need `libc`?

- "Bare-metal" Programming: Works (and technically enough), but not user-friendly
- Essential definitions that all programs use:
  - `stddef.h` – Provides types like `size_t`
  - `stdint.h` – Defines standard integer types like `int32_t`, `uint64_t`
  - `inttypes.h` – Defines formats for printing integer types

```
#include <stdio.h>
#include <stddef.h>
#include <stdint.h>

int main() {
   int64_t x = 1;
   printf("%ld\n", x); // %ld : this is a long = 4 bytes
         for 32-bit machine!
}
```

Why Do We Need `libc`?

- "Bare-metal" Programming: Works (and technically enough), but not user-friendly
- Essential definitions that all programs use:
  - `stddef.h` – Provides types like `size_t`
  - `stdint.h` – Defines standard integer types like `int32_t`, `uint64_t`
  - `inttypes.h` – Defines formats for printing integer types
  - `stdbool.h`
  - `float.h`
  - `limits.h`
  - `stdarg.h`
    - Used for handling variable arguments (essential in `syscall`, though custom `syscall0`, `syscall1`, etc., can be more efficient)

System Calls: The Minimal Interface of the OS

- System calls provide the OS's smallest, most compact interface.
- Not all system calls are as straightforward as `fork`; some require additional setup.

Low-Level API:

```
extern char **environ;
char *argv[] = { "echo", "hello", "world", NULL, };
if (execve(argv[0], argv, environ) < 0) {
  perror("exec");
}
```

High-Level API:

```
execlp("echo", "echo", "hello", "world", NULL);
system("echo␣hello␣world");
```

Comparison:

- Low-level APIs like `execve` offer more control but require more setup.
- High-level APIs like `execlp` and `system` simplify common tasks, making code easier to read and write.

Endless Abstractions: Layers of Encapsulation

- **Encapsulation (1): Pure Computations**
- **Encapsulation (2): File Descriptors**
- **Encapsulation (3): More Process / OS Functions**
- **Encapsulation (4): Address Space**

Address Space Management: `malloc` and `free`

Specification Overview (similar to Lab1):

- Manage a set of non-overlapping intervals $M = \{[\ell_0, r_0), [\ell_1, r_1), \ldots, [\ell_n, r_n)\}$ within a large interval $[L, R)$.

Operations

- `malloc(s)`:
  - Returns a segment of memory of size s.
  - May request additional memory from the OS if needed (observe this with `strace`).
  - Can "deny" requests if memory is insufficient.
- `free(`ℓ`, ` r`)`:
  - Given a starting address ℓ, removes the interval $[\ell, r) \in M$.

Considerations

- Inspired by concepts from *Introduction to Algorithms*.
- **Thread Safety**: Ensuring `malloc` and `free` work correctly in a multithreaded environment.
- **Scalability**: Handling multiple allocation and deallocation requests efficiently becomes a significant challenge.

Towards Efficient `malloc/free`

Premature optimization is the root of all evil.
— D. E. Knuth

Optimizing without workload analysis is risky

- **Workload Analysis:** Analyze common memory usage patterns to guide optimization.
- Key Principle: Allocating objects of size $O(n)$ should typically involve at least $\Omega(n)$ read/write operations. Otherwise, it's a performance bug.

Further Reading for Workload Analysis:

- Mimalloc: free list sharding in action (APLAS'19)

Towards Efficient `malloc/free`

Types of Memory Allocations:

- **Small Objects (high frequency):**
  - Strings, temporary objects (tens to hundreds of bytes), with varying lifespans.
- **Medium-Sized Objects (moderate frequency):**
  - Arrays, complex objects with longer lifespans.
- **Large Objects (low frequency):**
  - Huge containers, allocators, with very long lifespans.

Key Challenges:

- **Parallelism:** Allocations occur across all processors; parallel strategies are essential.
- **Data Structures:** Using linked lists or interval trees (e.g., first fit) is not ideal for high concurrency.

`malloc`: Fast and Slow Paths

Designing Two Systems for Memory Allocation:

- **Fast Path**
  - Optimized for high performance and parallelism.
  - Covers most allocation cases with minimal latency.
  - Has a small probability of failure, in which case it falls back to the slow path.
- **Slow Path**
  - Not focused on speed, but handles complex cases reliably.
  - Manages difficult allocations that cannot be handled by the fast path.

Common Pattern in Computer Systems:

- This fast-and-slow path design is common in systems, such as cache mechanisms or futex (as we discussed earlier).
- The fast path handles frequent, simple requests, while the slow path ensures robustness for exceptional cases.

`malloc`: Fast Path Design

Goal: Enable all CPUs to allocate memory in parallel

- **Thread-Local Allocation Buffer (TLAB):**
  - Each thread has its own "territory" or buffer for allocations.
  - By default, memory is allocated from its own buffer, minimizing contention.
- **Efficient Locking:**
  - Locks are rarely contested because threads typically allocate from their own buffer.
  - Only in rare cases, such as when memory is freed by another CPU, might a lock be required.
- **Global Pool Backup:**
  - When a thread's buffer runs low, it borrows memory from a global pool.
  - This approach allows for minor memory waste but improves allocation speed.
- **Alignment to 2^k Bytes:**
  - Aligning allocations to 2^k bytes helps maintain efficient memory access and reduces fragmentation.

Small Memory Allocation: Segregated List (Slab)

Allocation Strategy: Segregated List (Slab)

- Each **slab** contains objects of the same size.
- Each thread has its own slab for each object size.
- **Fast Path:** Allocation is completed immediately from the thread-local slab.
- **Slow Path:** Calls `pgalloc()` to allocate additional memory.

Two Implementation Approaches:

- **Global List:** A single global list of slabs.
- **List Sharding:** A small list per page, reducing contention.

Reclaiming Memory:

- Freed objects are returned directly to their respective slab.
- If the slab belongs to another thread, a **per-slab lock** is needed to prevent data races.

Endless Encapsulation

Moving Beyond C: Building on `libc`

- **C++ Compiler:** Expands on `libc` to support the C++ Standard Library.
- **OpenJDK (HotSpot):** Java runtime built on layers extending from C.
- **V8 (JavaScript):** A JavaScript engine that relies on foundational libraries for execution.
- **CPython:** The C-based Python interpreter extends further from `libc`.
- **Go:** Initially compiled with C, now capable of self-compilation. ("Goodbye, C!")

The Endless Cycle of Encapsulation:

- Each language and runtime builds upon lower-level abstractions, creating a layered stack.
- Over time, these layers form a "patched-up" world of interconnected technologies, making our computing environment both powerful and complex.

Excutable Linkable File (elf)

Making the Program Recognizable to the Machine

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

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.

Mapping Machine State to "C World" State:

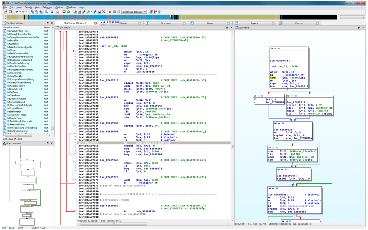
- The DWARF Debugging Standard (dwarfstd.org) defines an instruction set, `DW_OP_XXX`, that is Turing Complete.
- This instruction set can perform "arbitrary computations" to map the current machine state back to the C language state.

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

Reverse Engineering

- Provides insights into commercial software without access to the original source code.
- Challenges:
  - No Debug Information
  - Stripped Symbols
  - Opaque Instruction Sequences
- Techniques:
  - Analysts use specialized tools (e.g., `objdump`, `IDA Pro`, `Ghidra`) to disassemble and analyze the instruction sequences.
  - Techniques like pattern recognition, control flow analysis, and heuristic methods help infer program functionality.



Takeaways

- How the Shell Executes Programs?
- How to Build a Standard Library Above System Calls That Benefits Most Programs?
- What is an Executable File?
    - An executable file is a data structure describing a state machine.

Remember:

- RTFM (Read The "Fine" Manual), RTFSC (Read The File Source Code)
- Use the right tools: `binutils`, `gdb`.
- Performance optimization with fast/slow path design.