Lecture 19: Shared libraries

Brennon Brimhall

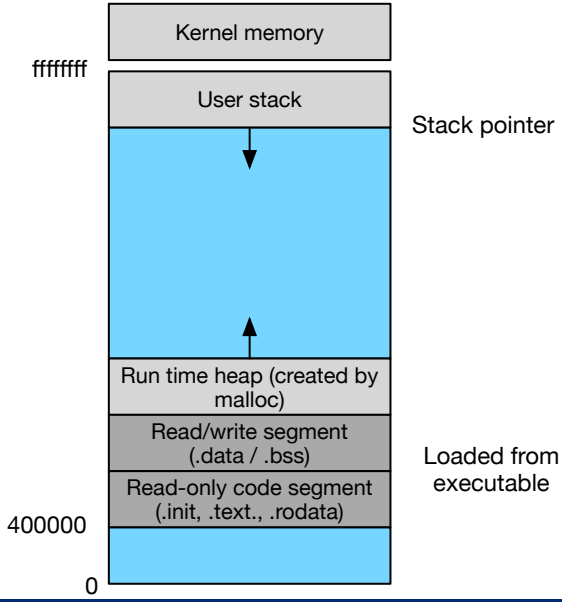
27 June 2025

Example code

- Example code for today is on the course website in `dynload.zip`

Shared libraries

Loading (Statically-linked) Executable Object Files



Dynamic Linking Shared Libraries

- Once program is executed, loader calls dynamic linker
- Dynamic linker "loads" shared library
- Nothing is actually loaded
- Memory mapping: pretend it's in memory
  (operating system deals with mapping of RAM address)
  - Much more about this when we cover virtual memory

Clicker quiz!

Clicker quiz omitted from public slides

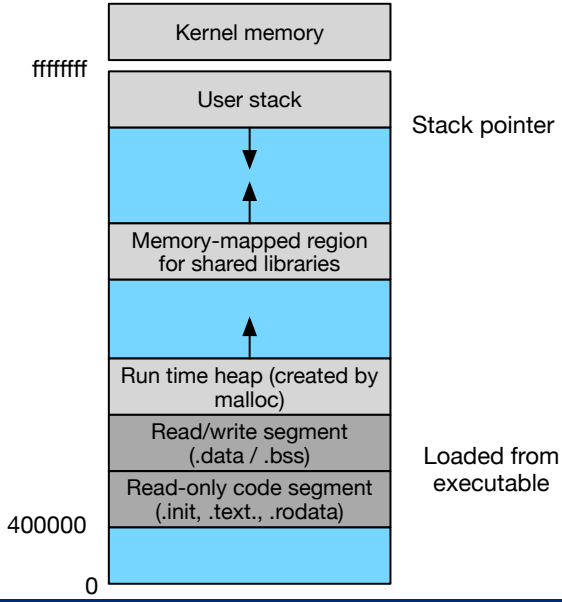
Clicker quiz!

Clicker quiz omitted from public slides

Example run

Clicker quiz omitted from public slides

Dynamic Linking Shared Libraries



Addresses in Shared Libraries

- Multiple processes use same shared library
- Idea: put it into a dedicated place in memory
- But
  - there may be many libraries
  - we may run out of address space
    (or at least waste it)
- Instead: compile into *position-independent code*

Position-Independent Code

- No matter where the libraries is loaded into memory
  $\rightarrow$ distances between addresses are the same
- Global offset table
  - table in data segment (relative position is known)
  - contains absolute addresses of variables and functions[1]
  - gets filled with correct values by dynamic linker
- Uses instruction point register (%rip)

Example

- Assume `b` is a global variable defined in a different executable object (e.g., a shared library), and our source code does `b++;`
- Global offset table (in data segment)

0	address of symbol a
1	address of symbol b
2	...

- Generated assembly code

```
mov 0x2008b9(%rip), %rax
addl $1, (%rax)
```

- Distance between code line and GOT entry 1 is 0x2008b9 bytes
- First line of code loads actual address of variable
- Second line increases it by 1

Filling in GOT entries: references to data

- When an executable object is loaded, the dynamic linker eagerly determines run-time addresses for all referenced global variables, and stores them in the appropriate GOT entries
- Each loaded executable object (including shared libraries) has a symbol table indicating relative offsets of code and data definitions
- The dynamic linker can use the symbol table information to determine the run-time address of any code or data definition
  - It knows the base address of each executable object, and the names and offsets of every symbol definition within it
  - So it's capable of producing an address for any defined symbol

Filling in GOT entries: references to code (functions)

- What about finding the addresses of called functions?
  - E.g., your code's `main` function calls `printf`, which is in the `libc` shared library
- This is way more complicated!
- *Very* brief explanation:
  - Function addresses are resolved *lazily*
  - Each (externally-defined) function has an entry in the GOT
  - Initial address in the GOT calls into the PLT (procedure linkage table), which invokes the dynamic loader to resolve the address of the called function and store it in the GOT
  - Subsequent calls load the function's address from the GOT

Tools for Manipulating Object Files

AR Creates static libraries, and inserts, deletes, and extracts members

STRINGS Lists all printable strings

STRIP Deletes symbol table information

NM Lists symbols defined in symbol table

READELF Displays complete structure

OBJDUMP Displays all information, useful to disassemble code

Dynamic loading of shared libraries

Shared libraries

- Shared libraries are loaded into memory at runtime
- Loading and symbol resolution happens automatically when an executable is linked against shared libraries
  - E.g., every C program is linked against the C library (`libc`), so the `libc` shared library is loaded automatically when the C program runs
- However, programs can also load shared libraries (and resolve symbols in them) *dynamically*

Very brief overview of dynamic loading on Linux

- `#include <dlfcn.h>`
- Call `dlopen` to dynamically load a shared library
- Once a shared library has been loaded, call `dlsym` to get the addresses of data and functions within it
- Call `dlclose` to unload the shared library
- Link the executable or shared library with `-ldl`

Function pointers

A function pointer is a pointer to a function. You can use them exactly as though they were functions.

Example:

```
// Declare a pointer to a function returning int and taking
// a single const char * parameter.
int (*ptr)(const char *);

// Make the function pointer point to a compatible function.
ptr = puts;

// Use the pointer to call the function it points to.
ptr("Hello world");
```

Why dynamic loading is useful

Scenarios where dynamic loading of shared libraries is useful include

- Interpositioning to "redefine" functions
- Extending program capabilities using "plugins"

Interpositioning

- Sometimes it is useful to "redefine" library functions
  - For debugging or tracing program behavior
  - To change or extend the behavior of the function
- This is sometimes referred to as "instrumenting" the executable in which functions are redefined by interpositioning
- "Interpositioning" means linking (statically or dynamically) functions with the same names into the executable
- Dynamic loading allows the interposed function(s) to call the "real" function(s)
- The `LD_PRELOAD` environment variable can be used to "inject" interposed definitions into an arbitrary program

A C program

Code:

```
// myprog.c
#include <stdio.h>

int main(void) {
  puts("Hello, world");
  return 0;
}
```

Compiling and running normally:

```
$ gcc -g -Wall -Wextra -pedantic -std=gnu99 -fPIC -c myprog.c -o myprog.o
$ gcc -o myprog myprog.o
$ ./myprog
Hello, world
```

Instrumenting the puts function

Code for an instrumented version of puts:

```
// instr.c
#include <stdlib.h>
#include <dlfcn.h>

int (*real_puts)(const char *s);

int puts(const char *s) {
  if (!real_puts) {
    void *handle = dlopen("/lib/x86_64-linux-gnu/libc.so.6", RTLD_LAZY);
    if (!handle) { exit(1); }
    *(void **) (&real_puts) = dlsym(handle, "puts");
    if (!real_puts) { exit(1); }
  }
  real_puts("This is the interposed version of puts!");
  return real_puts(s);
}
```

Instrumenting the program

Compiling the instrumentation library:

```
$ gcc -g -Wall -Wextra -pedantic -std=gnu99 -fPIC -c instr.c -o instr.o
$ gcc -shared -nostdlib -o instr.so instr.o -ldl
```

Using the instrumentation library:

```
$ ./myprog
Hello, world
$ LD_PRELOAD=./instr.so ./myprog
This is the interposed version of puts!
Hello, world
```

Plugins

- A “plugin” is a shared library intended to extend the functionality of a program
- Each plugin defines a standard set of functions
- “Host” program loads plugin share libraries dynamically, calls functions

Example plugin API

Example functions for plugins for an image-processing program:

```
const char *get_plugin_name(void);
const char *get_plugin_desc(void);
void *parse_arguments(int num_args, char *args[]);
struct Image *transform_image(struct Image *source, void *arg_data);
```

Each plugin defines its own versions of all of the required functions.

Host program loads plugin shared libraries and calls the functions as appropriate.

ABI

ABI = "Application Binary Interface"

For systems implemented using dynamic loading, the components must agree on the exact signatures of functions (parameters, return values) and the formats of common data structures (like `struct Image` in the previous example.)

Unexpected changes to function signatures or data layouts $\rightarrow$ plugins no longer work correctly.

These common specifications are called an "ABI".

Acknowledgements

Slides adapted from materials provided by David Hovemeyer.