2023 Fall ECE 344: Operating Systems
Lecture 3 1.0.0

Libraries

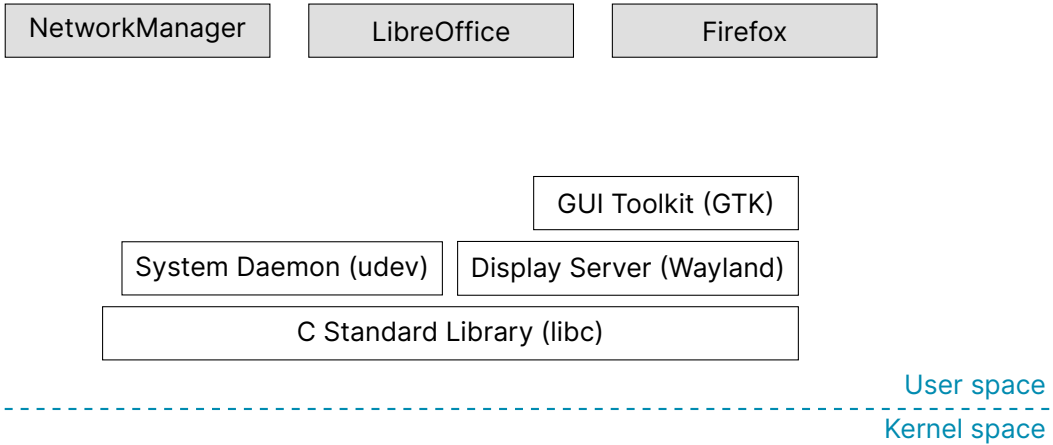
Jon Eyolfson
2023 Fall

What is an Operating System?

The kernel is part of an operating system, but what else?

Is macOS, iOS, iPadOS, watchOS, and tvOS all different operating systems?

Applications May Pass Through Multiple Layers of Libraries



What an Operating System is Depends on the Application

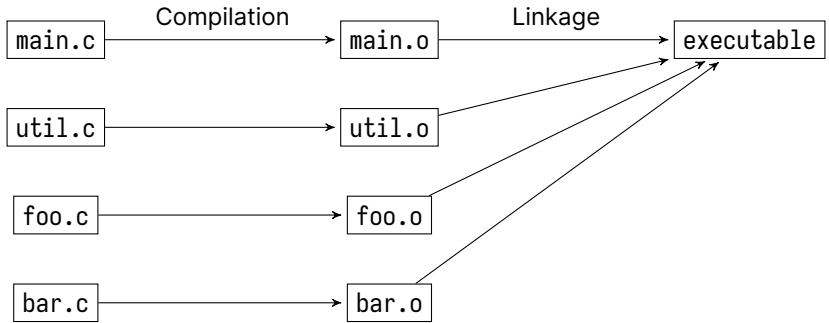
Android and Debian both use the Linux kernel, but the applications are different

Maybe they're the same OS if you only care about terminal applications

"Linux" distributions may be considered GNU/Linux
GNU distributes the standard C library and common utilities

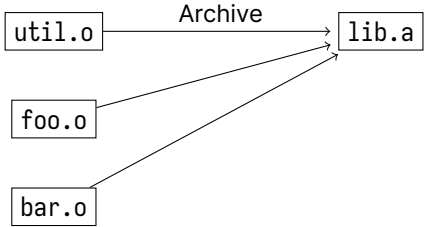
An operating system consists of a kernel and libraries required for your application

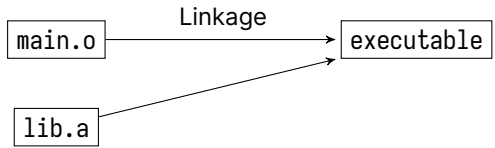
Normal Compilation in C



Note: object files (`.o`) are just ELF files with code for functions

Static Libraries Are Included At Link Time

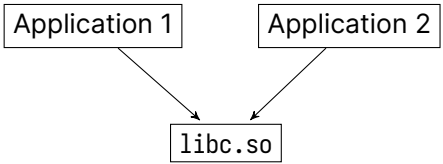




Dynamic Libraries Are For Reusable Code

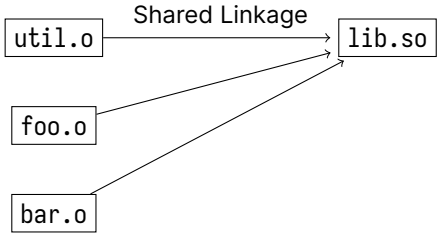
The C standard library is a dynamic library (`.so`), like any other on the system
    Basically a collection of `.o` files containing function definitions

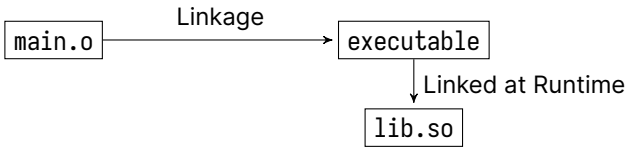
Multiple applications can use the same library:



The operating system only loads `libc.so` in memory once, and shares it

Dynamic Libraries Are Included At Runtime





Useful Command Line Utility for Dynamic Libraries

`ldd <executable>`

shows which dynamic libraries an executable uses

Static vs Dynamic Libraries

Another option is to statically link your code
Basically copies the `.o` files directly into the executable

The drawbacks compared to dynamic libraries:

- Statically linking prevents re-using libraries
  (commonly used libraries have many duplicates)
- Any updates to a static library requires the executable to be recompiled

What are issues with dynamic libraries?

Dynamic Libraries Updates Can Break Executables

A dynamic library update may subtly change the ABI causing a crash

Consider the following in a dynamic library:

A `struct` with multiple fields corresponding to a specific data layout (C ABI)

An executable accesses the fields of the `struct` used by a dynamic library

Now if a dynamic library reorders the fields

The executable uses the old offsets and is now wrong

Note: this is OK if the dynamic library never exposes the fields of a `struct`

C Uses a Consistent ABI for `structs`

`structs` are laid out in memory with the fields matching the declaration order
C compilers ensure the ABI of `structs` are the consistent for an architecture

Consider the following structures:

Library v1:

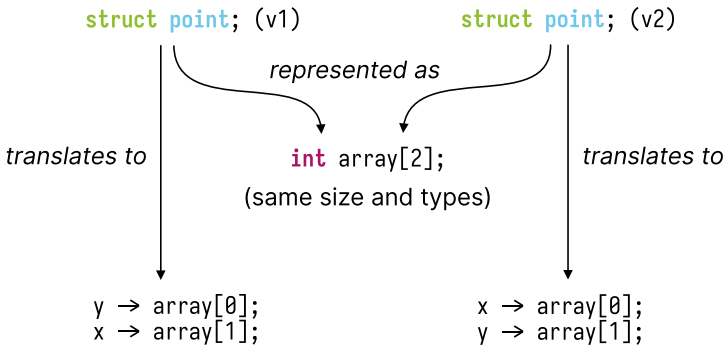
```
struct point {
  int y;
  int x;
};
```

Library v2:

```
struct point {
  int x;
  int y;
};
```

For v1 the `x` field is offset by 4 bytes from the start of `struct point`'s base
For v2 it is offset by 0 bytes, and this difference will cause problems

After Compilation the Translation Differs for Each Version



The Point API Has Four Functions

libpoint.so

```
struct point *point_create(int x, int y);
int point_get_x(struct point *p);
int point_get_y(struct point *p);
void point_destroy(struct point *p);
```

ABI Stable Code Should Always Print “1, 2” for Both Lines

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

#include "point.h"

int main(void) {
  struct point *p = point_create(1, 2);

  printf("point (x, y) = %d, %d (using library)\n",
         point_get_x(p), point_get_y(p));

  printf("point (x, y) = %d, %d (using struct)\n", p->x, p->y);

  point_destroy(p);
  return 0;
}
```

Try the Previous Example

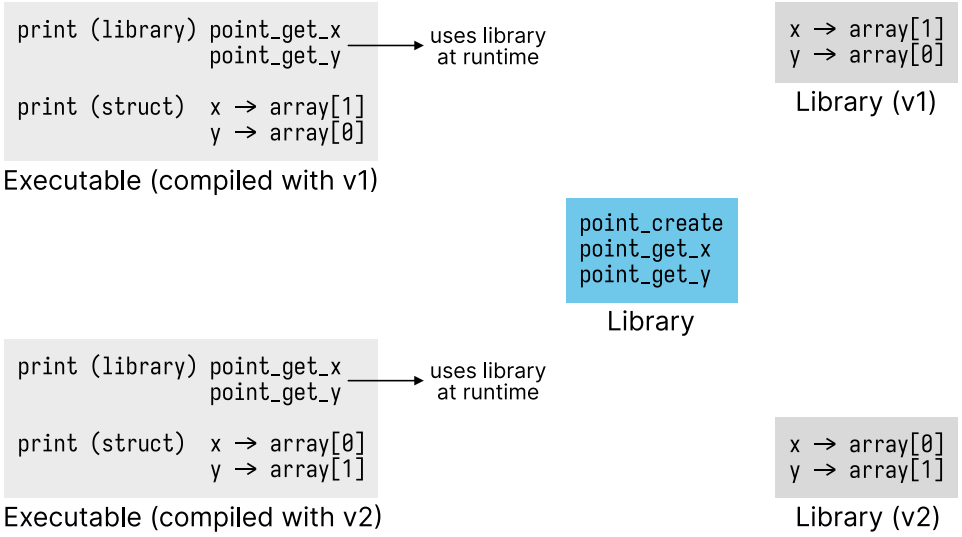
We could set `LD_LIBRARY_PATH` to `build/v1` or `build/v2` to simulate a library update

Run the following commands to see for yourself:

```
build/point-example-compile-v1-link-v1
LD_LIBRARY_PATH=build/v2 build/point-example-compile-v1-link-v1
```

Note: you'd also have a problem if you compiled with v2 and used v1

Mismatched Versions Don't Agree on the Location of X and Y



Mismatched Versions of This Library Causes Problems

The definition of `struct point` in both libraries is different
    Order of `x` and `y` change (and therefore their offsets)

Our code works correctly if the compiled and linked versions match
    If you expose a **`struct`** it becomes part of your ABI!

Let's try executing different combinations:

```
build/point-example-compile-v1-link-v1
build/point-example-compile-v1-link-v2
build/point-example-compile-v2-link-v1
build/point-example-compile-v2-link-v2
```

A proper stable ABI would hide the **`struct`** from `point.h`

Semantic Versioning Meets Developer's Expectations

From `https://semver.org/`

Given a version number MAJOR.MINOR.PATCH, increment the:

- MAJOR version when you make incompatible API/ABI changes
- MINOR version when you add functionality in a backwards-compatible manner
- PATCH version when you make backwards-compatible bug fixes

Dynamic Libraries Allow Easier Debugging

Control dynamic linking with environment variables
`LD_LIBRARY_PATH` and `LD_PRELOAD`

Consider the following example:

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

int main(void) {
  int *x = malloc(sizeof(int));
  printf("x = %p\n", (void *)x);
  free(x);
  return 0;
}
```

We Can Monitor All Allocations with Our Own Library

Normal runs of `alloc-example` outputs:

```
x = 0x561116384260
```

Create `liballoc-wrapper.so` that outputs all `malloc` and `free` calls

Run: `LD_PRELOAD=build/liballoc-wrapper.so build/alloc-example`

```
Call to malloc(4) = 0x55c12aa40260
Call to malloc(1024) = 0x55c12aa40280
x = 0x55c12aa40260
Call to free(0x55c12aa40260)
```

Interesting, we did not make 2 `malloc` calls

Detecting Memory Leaks with Valgrind

`valgrind` is another useful tool to detect memory leaks from `malloc` and `free`

Usage: `valgrind <executable>`

Here's a note from the `man` pages regarding what we saw:

"The GNU C library (`libc.so`), which is used by all programs, may allocate memory for its own uses. Usually it doesn't bother to free that memory when the program ends—there would be no point, since the Linux kernel reclaims all process resources when a process exits anyway, so it would just slow things down."

Note: this does not excuse you from not calling `free`!

Detecting Memory Leaks with AddressSanitizer

There's also sanitizer tools built into Clang (and now gcc), but you have to recompile

Add the `-Db_sanitize=address` flag to Meson

```
rm -rf build
meson setup build -Db_sanitize=address
meson compile -C build
```

System Calls are Rare in C

Mostly you'll be using functions from the C standard library instead

Most system calls have corresponding function calls in C, but may:

- Set `errno`
- Buffer reads and writes (reduce the number of system calls)
- Simplify interfaces (function combines two system calls)
- Add new features

C `exit` Has Additional Features

System call `exit` (or `exit_group`): the program stops at that point

C `exit`: there's a feature to register functions to call on program exit (`atexit`)

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

void fini(void) {
  puts("Do fini");
}

int main(void) {
  atexit(fini);
  puts("Do main");
  return 0;
}
```

Operating Systems Provide the Foundation for Libraries

We learned:

- Dynamic libraries and a comparison to static libraries
  - How to manipulate the dynamic loader
- Example of issues from ABI changes without API changes