CS 111: Operating System Principles

Lecture 3

# Libraries

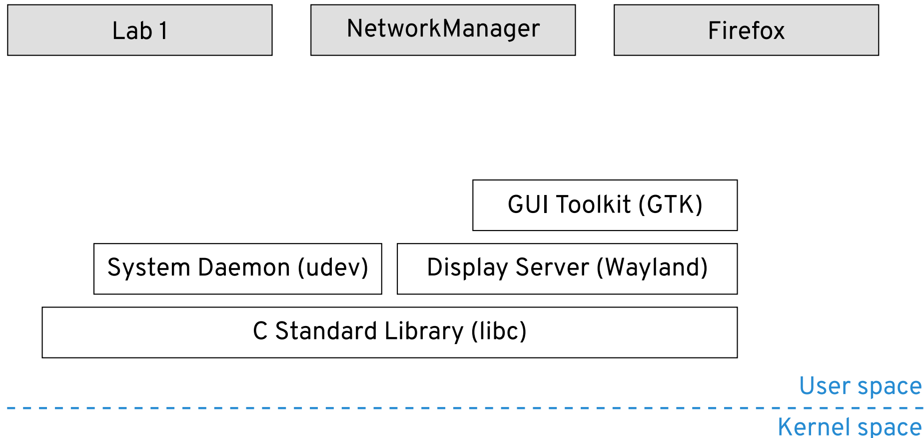
2.0.1

Jon Eyolfson

June 29, 2021

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License

# Applications May Pass Through Multiple Layers of Libraries



# C ABI, or Calling Convention for x86-64

System calls use registers, while C is stack based:

- Arguments pushed on the stack from right-to-left order
- `rax`, `rcx`, `rdx` are caller saved
- Remaining registers are callee saved

What advantages does this give us vs system call ABI? Disadvantages?

# 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`)

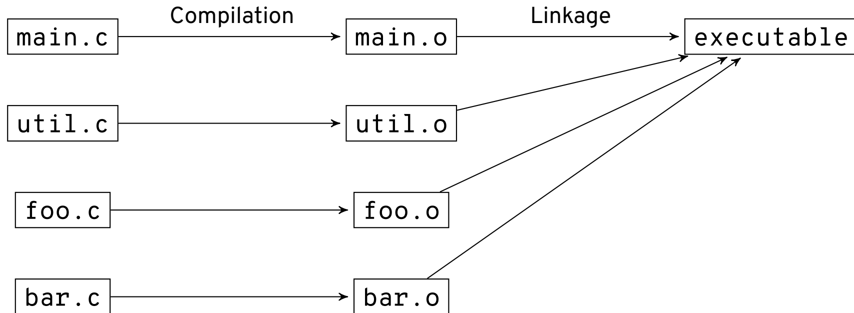
```
#include <stdio.h>
#include <stdlib.h>

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

int main(int argc, char **argv) {
  atexit(fini);
  puts("Do main");
  return 0;
}
```

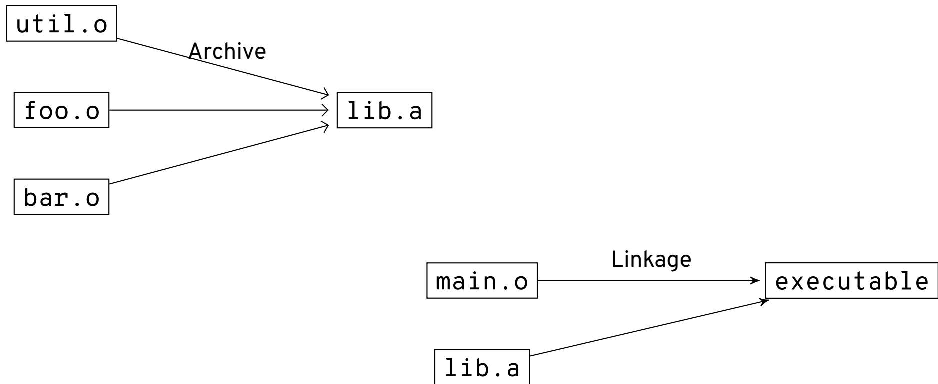
`examples/lecture-03/atexit-example`

# Normal Compilation in C




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

# Static Libraries Are Included At Link Time

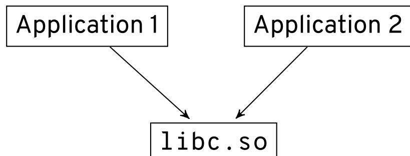


# Dynamic Libraries Are For Reusable Code

The C standard library is a dynamic library (`.so`), like any other on the system

- Basically a collection of `.o` files containing function definitions

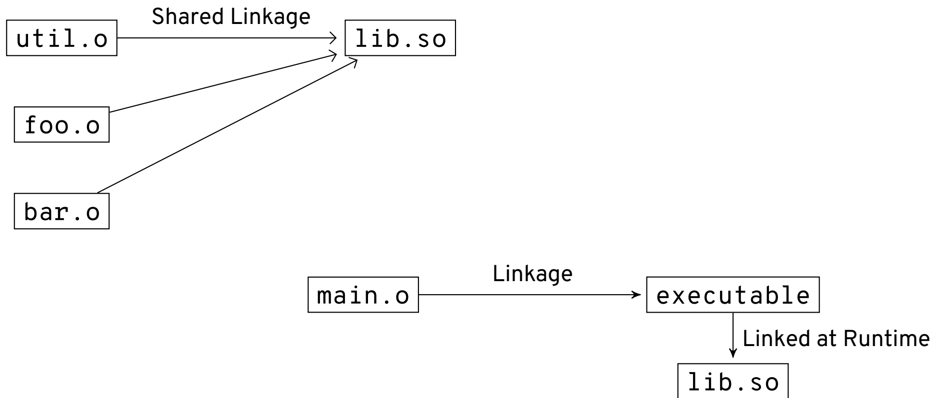
Multiple applications can use the same library:




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

- The same physical page corresponds to different virtual pages in processes

# Dynamic Libraries Are Included At Runtime



# Useful Command Line Utilities for Dynamic Libraries

```
ldd <executable>
```

shows which dynamic libraries an executable uses

```
objdump -T <library>
```

shows the symbols (often just function names) that are in the library

You can also use `objdump -d` to disassemble the library

# 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 with ABI Changes

An update to a dynamic library can easily cause an executable using it to 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` in the dynamic library

Now if the dynamic libraries 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

# ABI Stable Code Should Always Print “1, 2”

```
int main(int argc, char **argv) {
  struct point *p = point_create(1, 2);

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

  struct point_v1 *p_v1 = (struct point_v1 *) p;
  printf("point (x, y) = %d, %d (using v1)\n", p_v1->x, p_v1->y);

  struct point_v2 *p_v2 = (struct point_v2 *) p;
  printf("point (x, y) = %d, %d (using v2)\n", p_v2->x, p_v2->y);

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

# Mismatched Versions of This Library Causes Unexpected Results

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

- Our code works correctly with either v1 or v2 of the library
  - The stable ABI is in `libpoint.h` (it hides the `struct`)
    - If you expose a `struct` it becomes part of your ABI!

- If the `struct point` was exposed we get unexpected results with v2
  - This would be compiled into your program if the `struct` was visible!

# Try the Previous Example

It's in `examples/lecture-03` directory

Set `LD_LIBRARY_PATH` to `lib-v1` or `lib-v2` to simulate a library update

Run the following commands to see for yourself:

```
LD_LIBRARY_PATH=lib-v1 ./point-example
LD_LIBRARY_PATH=lib-v2 ./point-example
```

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

# 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:

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

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

# We Can Monitor All Allocations with Our Own Library

Normal runs of `alloc-example` outputs:

```
x = 0x561116384260
```

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

Run: `LD_PRELOAD=./alloc-wrapper.so ./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

`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`!

# Standard File Descriptors for Unix

All command line executables use the following standard for file descriptors:

- `0` – `stdin` (Standard input)
- `1` – `stdout` (Standard output)
- `2` – `stderr` (Standard error)

The terminal emulators job is to:

- Translate key presses to bytes and write to `stdin`
- Display bytes read from `stdout` and `stderr`
- May redirect file descriptors between processes

# Checking Open File Descriptors on Linux

`/proc/<PID>/fd` is a directory containing all open file descriptors for a process

`ps x` command shows a list of processes matching your user (lots of other flags)

A terminal emulator may give the output:

```
> ls -l /proc/21151/fd
0 -> /dev/tty1
1 -> /dev/tty1
2 -> /dev/tty1
```

`lsof <file>` shows you what processes have the file open

For example, processes using C: `lsof /lib/libc.so.6`

# 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
- Standard file descriptor conventions for UNIX