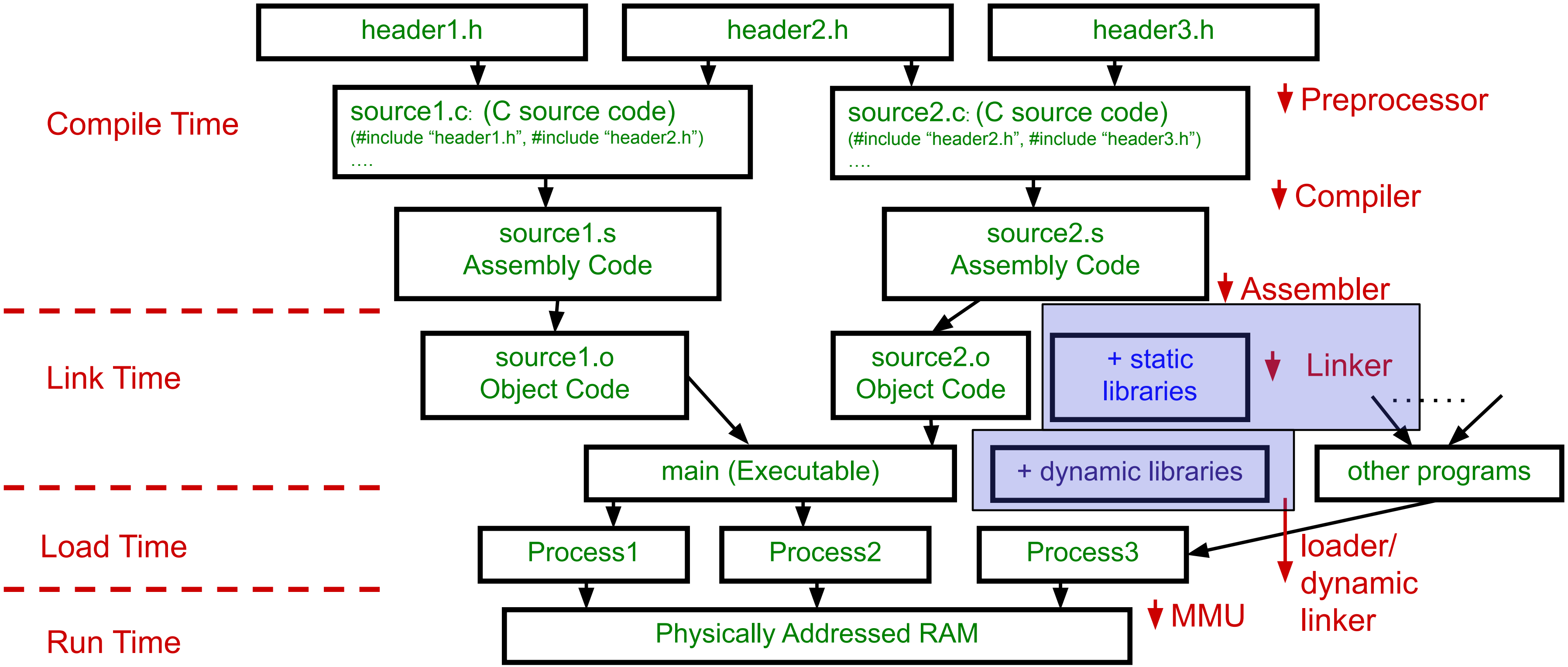
CS 3214: Computer Systems
Lecture 13: Linking + Loading (3)

Instructor: Huaicheng Li

Oct 4 2022

VIRGINIA TECH™



Compile Time

Link Time

Load Time

Run Time

header1.h

header2.h

header3.h

source1.c: (C source code)
(#include “header1.h”, #include “header2.h”)
….

source2.c: (C source code)
(#include “header2.h”, #include “header3.h”)
….

Preprocessor

Compiler

source1.s
Assembly Code

source2.s
Assembly Code

Assembler

source1.o
Object Code

source2.o
Object Code

+ static
libraries

Linker

main (Executable)

+ dynamic libraries

other programs

Process1

Process2

Process3

loader/
dynamic
linker

Physically Addressed RAM

MMU

Packing Common Functions

- ❑ Such as math, string manipulation, etc.
- ❑ Write your own "common.c" file, compile it to an object file, and link it to programs that will use it
  - ▪ Time and space inefficient
- ❑ One function in each .c file, compile all of them, and choose to link the ones that's needed
  - ▪ Need to know exactly which one to use/link
  - ▪ Burdens on programmers to maintain so many object files

Static Libraries

- ❑ Pack multiple relocatable object files into a single file with an index (a.k.a, archive)
  - ".a" archive files
  - "*ar rs mylib.a a.o b.o c.o* …"
- ❑ Example libraries
  - "libc.a" → C standard library (e.g., /usr/lib/x86_64-linux-gnu/libc.a)
  - "libm.a" → C math library
- ❑ "ar –t libc.a" → Check all the object files in the library
- ❑ "nm –s libc.a" → Check all the symbols in the library

Static Libraries

- ❑ **Static Libraries:** Pack multiple relocatable object files into a single file with an index (a.k.a, archive)
  - ▪ ".a" archive files
  - ▪ "*ar rs mylib.a a.o b.o c.o …*"
  - ▪ "libc.a" → C standard library, "libm.a" → C math library
    - "ar –t libc.a" → Check all the object files in the library

- ❑ **How does linker resolve dependency?**
  - ▪ Scan ".o" and ".a" file in the order specified in the command line
  - ▪ Keep a list of unresolved symbols
  - ▪ For each ".o" or ".a" file, try to resolve unresolved symbols
    - If an archive member (.o file) resolves the dependency, link it
  - ▪ If still unresolved symbols at the end, error
  - ▪ When processing a library, the linker will include a .o module from this library if and only if it defines a symbol that is currently in set U

Static Libraries

- **Pros:**
  - Only needed .o files are included/linked
  - Override a library symbol by specifying a definition in a library that will be listed first
  - Compatibility (w/ all dependencies included)
- **Cons:**
  - Linking behavior depends on the exact order in which .o files and libraries are listed on the command line
  - May be necessary to list libraries in a certain order (-lXm –lXt –lX11), or multiple times if they have mutual dependencies, or use special linker grouping option (--start-group/--end-group)
  - Duplication in the executable
  - Updates on libraries requires applications to relink
  - Error prone and confusing (but, linker maps help track down how the linker resolved symbols)
  - Larger size (executable file size, and requires more memory when loaded)
  - No Sharing

Shared Libraries

- ❑ Object files that contain code and data are loaded and linked into an application dynamically, at
  - ▪ load-time
  - ▪ run-time
- ❑ **Linux: ".so" files**, Windows: Dynamic link libraries (DLLs)
- ❑ **Can be shared by multiple processes**
  - ▪ Mapped into different virtual addresses within different processes
  - ▪ Memory must be read-only and content not be dependent on the position at which it is mapped
- ❑ **Load-time linking**
  - ▪ Common case in Linux, handled automatically by the dynamic linker (ld-linux.so)
  - ▪ Standard C library (libc.so) usually dynamically linked
  - ▪ Executable still contains external references that will be resolved at load-time
  - ▪ Recursive: a dynamically linked library may have other dependencies
- ❑ **Run-time linking**
  - ▪ dlopen() interface
- ❑ Semantics almost the same as static libraries

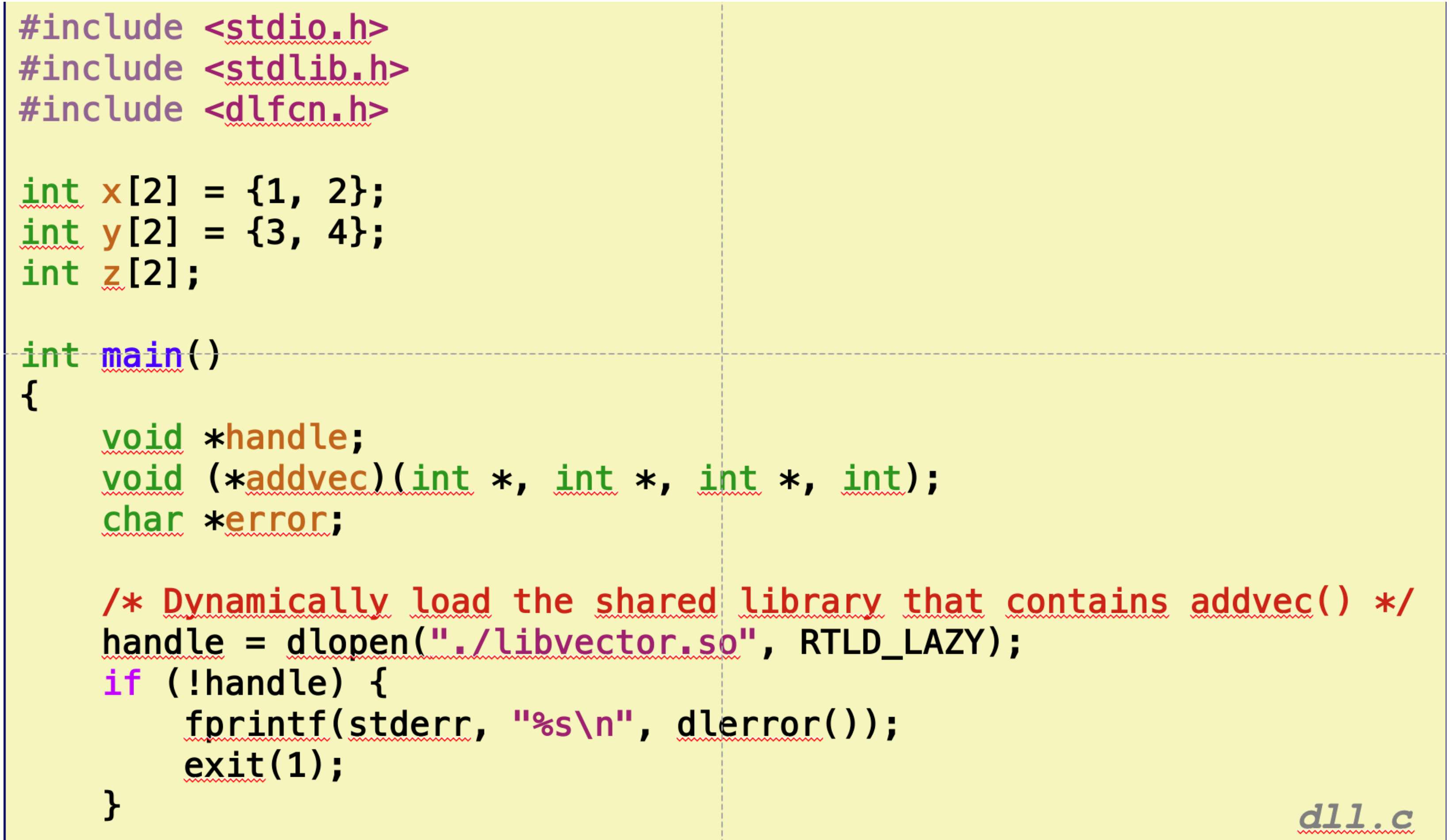
Implementation of Shared Libraries

- ❑ Position-Independent Code (handles intra-library references)
  - X86_64: PC-relative addressing mode ($rip + offset)
- ❑ If a library defines global function f or variable x, the address f and &x are not known until the library is loaded
  - Indirect function calls (via entries in PLT (Procedure Linkage Table))
  - On-demand loading via trampolines: first access trigger jump into dynamic linker
  - Subsequent jumps go straight to loaded function
- ❑ In general ,sahred libraries introduce a marginal cost at runtime

Dynamic Linking at Load-time

- ❑ "gcc –shared –o liba.so a.c b.c"
- ❑ "gcc –c main.c –o main.o"
- ❑ Linker (ld) on main.o and liba.so → Partially linked executable object file
  - ▪ Relocation and symbol table information from .so file
- ❑ Load executable binary (execve()) and .so into fully linked executable in memory
  - ▪ Need code and data from .so file

Dynamic Linking at Run-time



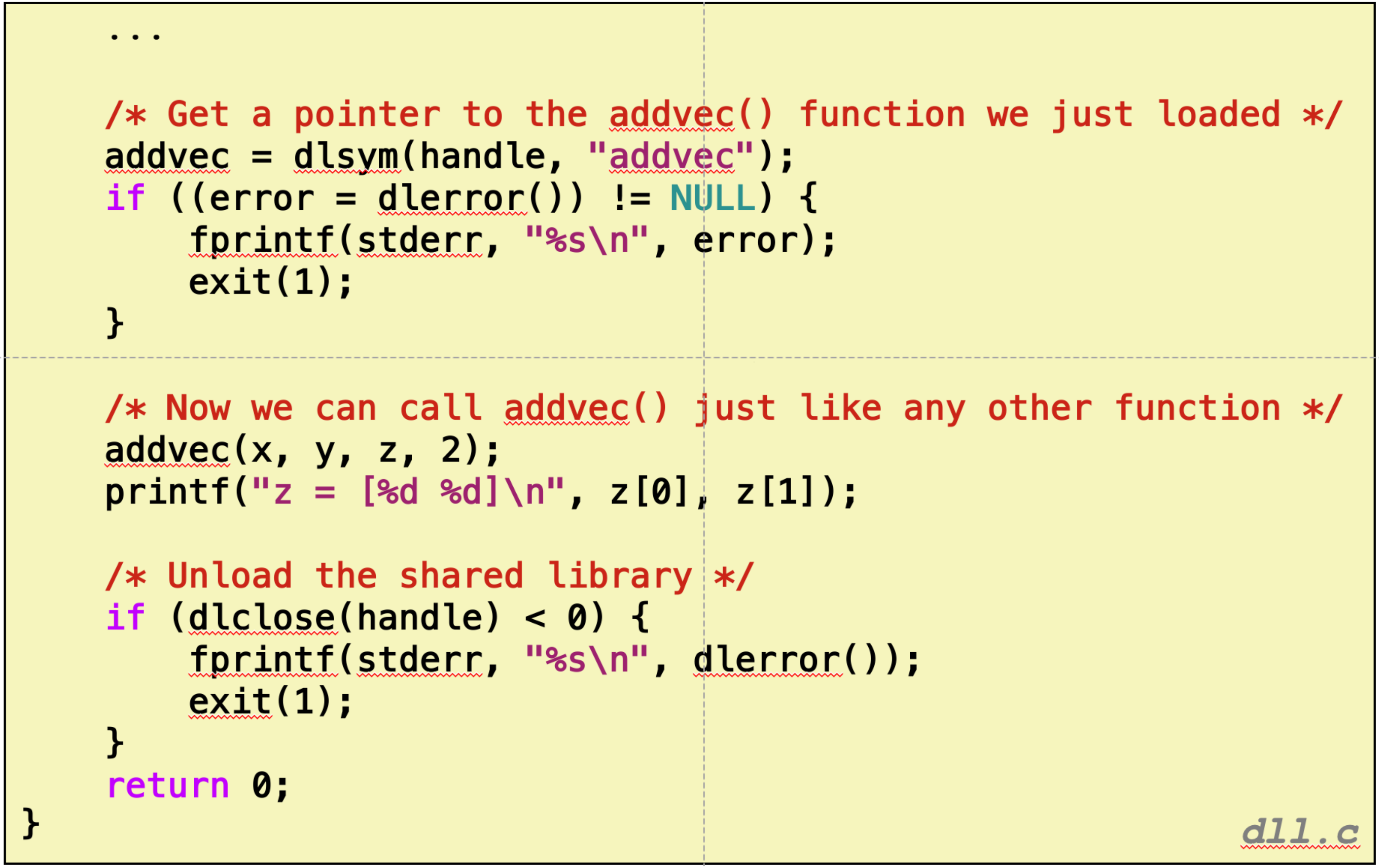
```
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

int x[2] = {1, 2};
int y[2] = {3, 4};
int z[2];

int main()
{
    void *handle;
    void (*addvec)(int *, int *, int *, int);
    char *error;

    /* Dynamically load the shared library that contains addvec() */
    handle = dlopen("./libvector.so", RTLD_LAZY);
    if (!handle) {
        fprintf(stderr, "%s\n", dlerror());
        exit(1);
    }
```

dll.c



```
    ...

    /* Get a pointer to the addvec() function we just loaded */
    addvec = dlsym(handle, "addvec");
    if ((error = dlerror()) != NULL) {
        fprintf(stderr, "%s\n", error);
        exit(1);
    }

    /* Now we can call addvec() just like any other function */
    addvec(x, y, z, 2);
    printf("z = [%d %d]\n", z[0], z[1]);

    /* Unload the shared library */
    if (dlclose(handle) < 0) {
        fprintf(stderr, "%s\n", dlerror());
        exit(1);
    }
    return 0;
}
```

dll.c

Library interpositioning

- ❑ Intercept calls to arbitrary functions
  - ▪ Compile-time
    - – Macro expanded into self-defined function calls
  - ▪ Link-time (LD_PRELOAD)
  - ▪ Run-time