Week 8
Dynamic Linking

19 November 2018

CS 35L Lab 4

Jeremy Rotman

Announcements

- ➔ Assignment #7 is due Monday by 11:55pm
- ➔ Assignment #10 Presentations
  - ◆ **Email me to tell me what story you are choosing**
  - ◆ Here is the link to see what stories people have signed up for already
- ➔ My Office Hours this week will be cancelled
  - ◆ In case anyone was planning on being at office hours Thursday
    - If you were, sorry but I don't intend to be on campus Thursday
- ➔ If you won't be here Wednesday

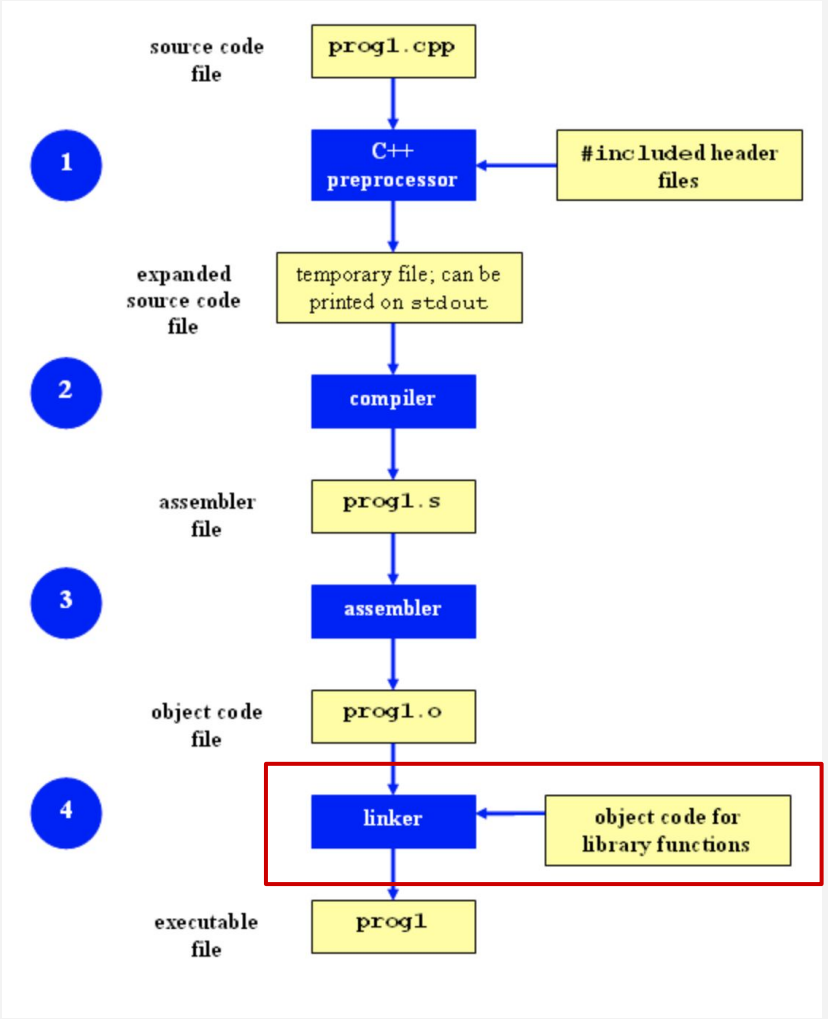
Outline

- ➔ Static and Dynamic Linking
- ➔ Implementing Dynamic Loading
- ➔ Entropy Pools
- ➔ Assignment 7

Questions?

C++ Compilation Process

1. C++ preprocessor copies header, generates macro, and checks defined constants
2. Source code compiled to assembly
3. Assembler code is assembled into object code
4. Object code is linked with object code files for library functions, producing an executable



Linking

➔ What does the linker actually do?

Linking

- ➔ What does the linker actually do?
  - The linker collects procedures and links together the object modules into one executable program
- ➔ Why use linking instead of a single huge program?
  - Efficiency
    - Only need to recompile the external functions
    - Don’t need to recompile the entire huge program

Static Linking

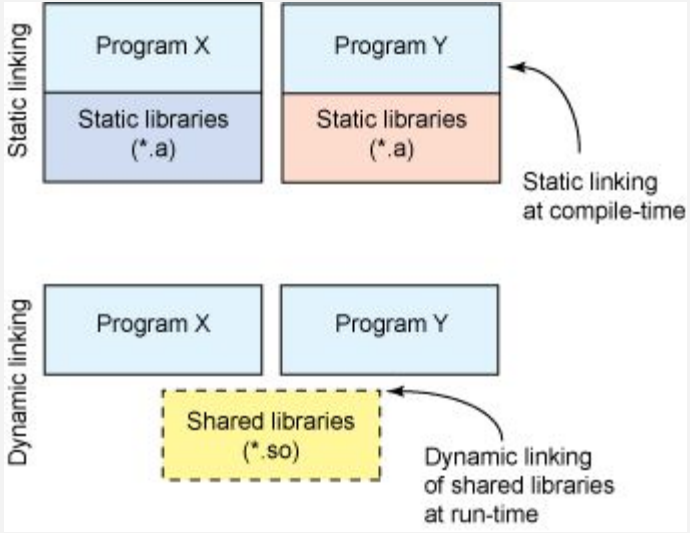
- ➔ Carried out once to produce an executable file
- ➔ If static libraries are called, the linker copies all modules referenced by the program to the executable
- ➔ Typically denoted by the .a extension

Dynamic Linking

- ➔ Allows a process to add, remove, replace or relocate object modules during execution
- ➔ If shared libraries are called
  - ◆ Only copy a little reference information when the executable file is created
  - ◆ Complete the linking during loading time or running time
- ➔ Typically denoted by the .so file extension
  - ◆ .dll on windows

Static vs. Dynamic Linking

- Static Linking
  - Each program has its own complete copy of the library
- Dynamic Linking
  - Each program shares a reference to the shared library
  - The library is loaded at execution



Dynamic Loading

- The Dynamic Loading (DL) API
  - Allows the application to specify a library to load
  - Then use the functions within that library from the executable
- Libraries are still linked with dynamic linking

DL API

Function	Description
dlopen	Makes an object file accessible to a program
dlsym	Obtains the address of a symbol within a `dlopen`ed object file
dlerror	Returns a string error of the last error that occurred
dlclose	Closes an object file

DL API

- include <dlfcn.h>
- void* dlopen(const char* *filename*, int *flag*)
  - *filename* is the name of the library file
  - *flag* is one of either RTLD_LAZY or RTLD_NOW
    - RTLD_LAZY: lazy linking, symbols only resolved when executed
    - RTLD_NOW: All symbols resolved before dlopen returns
- void* dlsym(void* *handle*, const char* *symbol*)
  - Takes *handle* (from dlopen) and returns the address in memory where symbol is stored
    - Here *symbol* would be the name of the function you need

DL API

- ➔ int dlclose(void)
  - Decrements the handle’s reference count
  - If the reference count reaches 0, handle is unloaded
  - Returns 0 on success

Example

```
    void *handle;
    double (*cosine)(double);
    char *error;

   handle = dlopen("libm.so", RTLD_LAZY);
    if (!handle) {
        fprintf(stderr, "%s\n", dlerror());
        exit(EXIT_FAILURE);
    }

   dlerror();    /* Clear any existing error */

   /* Writing: cosine = (double (*)(double)) dlsym(handle, "cos");
       would seem more natural, but the C99 standard leaves
       casting from "void *" to a function pointer undefined.
       The assignment used below is the POSIX.1-2003 (Technical
       Corrigendum 1) workaround; see the Rationale for the
       POSIX specification of dlsym(). */

   *(void **) (&cosine) = dlsym(handle, "cos");

   if ((error = dlerror()) != NULL)  {
        fprintf(stderr, "%s\n", error);
        exit(EXIT_FAILURE);
    }

   printf("%f\n", (*cosine)(2.0));
    dlclose(handle);
    exit(EXIT_SUCCESS);
}
```

- dlerror() returns a human readable error string
- dlerror's return value is checked after dlsym because NULL is a valid symbol to be returned by dlsym.

Advantages and Disadvantages of Dynamic Linking

- Advantages
  - The executable is smaller
  - If the library changes, the code referencing it does not need to recompile
  - Multiple programs can access the .so file at the same time
- Disadvantages
  - Performance hit
    - Objects need to be loaded, addresses need to be resolved, etc.
  - Missing libraries
  - Varying versions of libraries

GCC Flags

- -fPIC
  - Compiler directive to generate Position Independent Code
- -l*library*
  - Searches for the file lib*library.a*
  - If not given -L, it defaults to the /usr/lib path
- -L
  - At compile time, search for the library from this path
- -Wl,*option*
  - passes *option* to the linker
  - For multiple options use more commas

GCC Flags

- -rpath=.
  - At runtime, find .so from this path
- -c
  - Generate object code from c code
- -shared
  - Produce a shared object which can then be linked with other objects to form an executable

Attributes of Functions

- ➔ Used to declare certain things about functions called in your program
- ➔ Also used to control memory placement, code generation options or call/return conventions within the function being annotated
- ➔ Introduced by the attribute keyword on a declaration, followed by an attribute specification inside double parentheses

Attributes of Functions

- __attribute__((constructor))
  - Will make the function run before main
  - Runs when dlopen is called
- __attribute__((destructor))
  - Will make the function run after main
  - Runs when dlclose() is called
- E.g.

```
__attribute__((constructor))
void run_before(void) {
    printf("called first\n");
}
```

Entropy Pool

- Entropy is randomness collected by an OS
- Every time you draw a random number from the OS, some entropy is depleted from the entropy pool
- Entropy is slowly built by the OS
- Why is this an issue?
  - The homework requires you to generate random numbers
  - If enough students are working at the same time:
    - Entropy might be depleted faster than it is generated
  - If there is not enough entropy, randint will fail
- Solution: If you are running into depleted entropy, trying logging onto a different server

Lab 7

- Implement a simple C program
  - Computes cos(sqrt(3.0))
- Check what is loaded using the **ldd** command
- Use **strace** to see what system calls are made related to linking
  - Hint: It has to open library files and link them to memory
- Run ls /usr/bin | awk ‘NR%101==*nnnnnnnn*%101’
  - *nnnnnnnn* is you UID
  - This generates ~24 commands to investigate with LDD
- Make a sorted list of all dynamic libraries you find
  - Remove duplicates

Homework 7

- Split randall.c into 4 files
- Stitch the files together with static and dynamic linking
- randmain.c must use dynamic loading to dynamically link with randlibhw.c and randlibsw.c
- Write randmain.mk (makefile include file) to do the linking
  - When make encounters an include statement
    - suspends reading from current makefile
    - read from each of the listed files
    - When finished proceed in the containing makefile

Homework 7

- ➔ randall.c outputs N random bytes of data
  - ◆ Helper functions to check if hardware random generator is available
    - If so, generates random number from hardware
    - Hardware RNG exists if RDRAND instruction exists
    - Uses cpuid to check whether CPU supports RDRAND
      - 30th bit of ECX register is set
  - ◆ Helper functions to check if random numbers using a software implementation
    - /dev/urandom
  - ◆ Main
    - Gets input *N*
    - Runs either hardware RNG or software RNG to get *N* random bytes

Homework 7

- Your goal is to divide randall.c into linked modules and a main program
- randcpuid.c
  - Code to determine whether CPU has RDRAND instruction
  - Include and implement interface in randcpuid.h
- randlibhw.c
  - Hardware implementation of RNG
  - Include and implement interface in randlib.h
- randlibsw.c
  - Software implementation of RNG
  - Include and implement interface in randlib.h

Homework 7

- randmain.c
  - Contains the main program that glues together everything else
  - Includes randcpuid.c (linked statically)
  - Does not include randlib.h (linked during runtime)
  - Depending on whether RDRAND is supported
    - Dynamically load the hardware or software implementation of randlib.h

Questions?