CS 35L- Software Construction Laboratory

Winter 2019

Week 8

TA: Guangyu Zhou

Announcement

- Course schedule for the last 3 weeks
  - This week: Finishing Dynamic linking
  - Week of 03-04 & 03-06 : SSH (Bring your Beagle Bones next class!!)
  - Week of 03-11 & 03-13: Change management + Final review
  - **Final exam: March 17 (Sunday)** from 3:00-6:00pm
    - Check back on 03/04/2019 (next Monday) for final exam location

Review

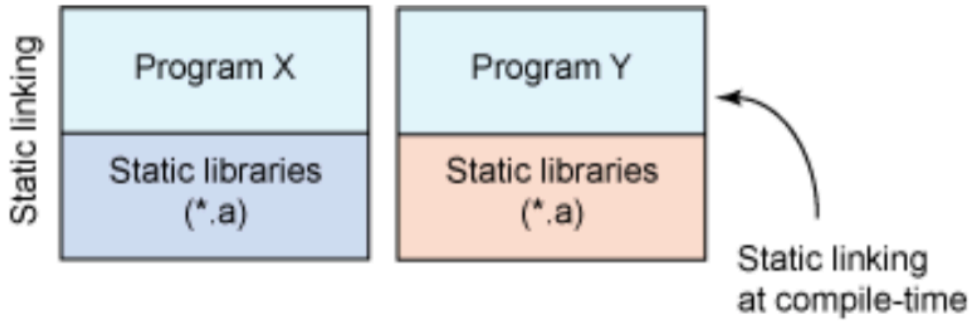
- static linking
  - Consumes more space
  - **More portable, can execute on platform with no standard library installed**

- dynamic linking
  - Consumes less space
  - Less portable, cannot execute on platform with no standard library installed
  - **Easy to update and fix bugs in standard library**

Anatomy of Linux shared libraries

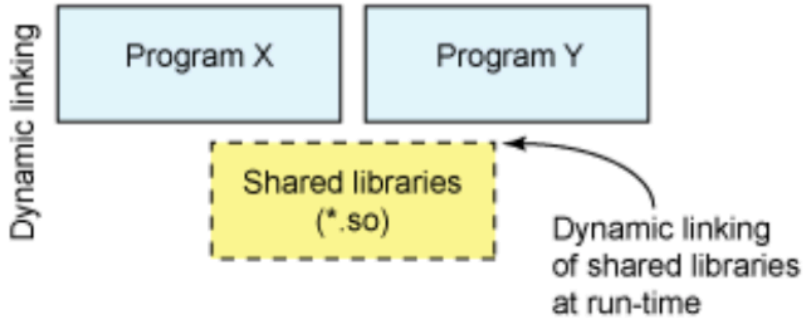
static library

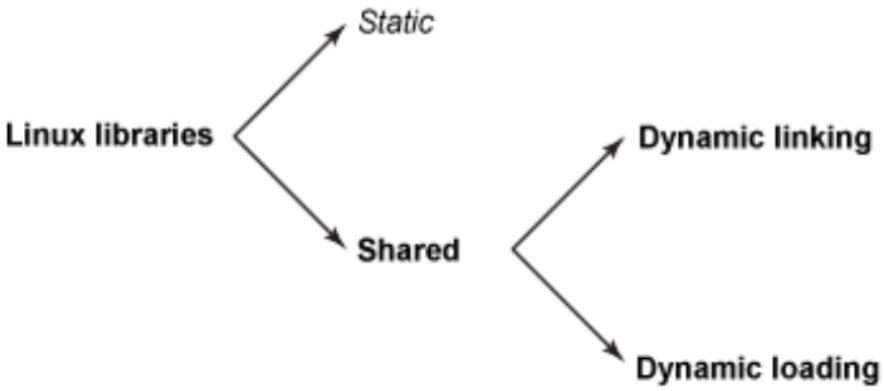
functionality to bind to a program statically at compile-time



dynamic library

functionality to bind to a program dynamically at run-time





dynamic linking - have Linux load the library upon execution

dynamic loading - selectively call functions with the library in a process

Dynamic Loading

- Let an application load and link libraries itself
  - application can specify a particular library to load, then
  - application can call functions within that library
- Load shared libraries from disk (file) into memory and re-adjust its location
- The Dynamic Loading API

Table 1. The DI 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

src: https://www.ibm.com/developerworks/library/l-dynamic-libraries/

The Dynamic Loading API

- dlopen - makes an object file accessible to a program
  - void *dlopen(const char *file, int mode);
  - RTLD NOW → relocate now; RTLD LAZY → to relocate when needed;
- dlsym - gives resolved address to a symbol within this object
  - void *dlsym(void *restrict handle, const char *restrict name);
  - check char *dlerror(); if an error occurs
- dlerror - returns a string error of the last error that occurred
- dlclose - closes an object file

Dynamic Loading: Example

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

int main(int argc, char* argv[]) {
 int i = 10;
 void (*myfunc)(int *); void *dl_handle;
 char *error;

 dl_handle = dlopen("libmymath.so", RTLD_LAZY);//RTLD_NOW

 if(!dl_handle) {
  printf("dlopen() error - %s\n", dlerror()); return 1;
 }
 //Calling mul5(&i);
 myfunc = dlsym(dl_handle, "mul5"); error = dlerror();
 if(error != NULL) {
  printf("dlsym mul5 error - %s\n", error); return 1;
 }
 myfunc(&i);
  //Calling add1(&i);
  myfunc = dlsym(dl_handle, "add1"); error = dlerror();
  if(error != NULL) {
   printf("dlsym add1 error - %s\n", error); return 1;
  }
  myfunc(&i);
  printf("i = %d\n", i);
  dlclose(dl_handle);
  return 0;
}
```

- Copy the code into main.c
- gcc main.c -o main –ldl
- You need to set the environment variable **LD_LIBRARY_PATH** to include the path that contains libmymath.so

Create static and shared libs in GCC

- mymath.h

```
#ifndef _ MY_MATH_H
#define _ MY_MATH_H
void mul5(int *i);
void add1(int *i);
#endif
```

- mul5.c

```
#include "mymath.h"
void mul5(int *i)
{
  *i *= 5;
}
```

- add1.c

```
#include "mymath.h"
void add1(int *i)
{
  *i += 1;
}
```

- gcc -c mul5.c -o mul5.o
- gcc -c add1.c -o add1.o
- ar -cv libmymath.a mul5.o add1.o → (static lib)
- gcc -shared -fpic -o libmymath.so mul5.o add1.o → (shared lib)

Attributes of Functions

- Used to declare certain things about functions called in your program
  - Help the compiler optimize calls and check code
- 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

Reference: https://gcc.gnu.org/onlinedocs/gcc-3.1/gcc/Function-Attributes.html

Attributes of Functions

- `__attribute__ ((__constructor__))`
  - Is run when `dlopen()` is called
- `__attribute__ ((__destructor__))`
  - Is run when `dlclose()` is called
- Example:

```
__attribute__ ((__constructor__))
void to_run_before (void) {
    printf("pre_func\n");
}
```

Review of Makefile

- Makefile:
  - Tool (Programming language) to build executable program from (C/C++) source code automatically
  - Utility for managing large software projects
  - Compiles files and keeps them up-to-date
  - Efficient Compilation (only files that need to be recompiled)

Review of Makefile Example

- Syntax of Makefile:
  - A single Makefile is built with multiple rules
  - Syntax for each rule:

    target: prerequisites

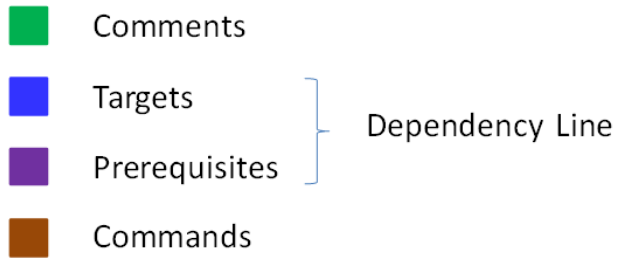
    [tab] bash command

    [tab] bash command

    ....
  - target and prerequisites are normally files
  - During execution, if target does not exist or modification time of any prerequisites is newer than the target, execute the bash command.

```
# Makefile - A Basic Example
all : shop  #usually first
shop : item.o shoppingList.o shop.o
        g++ -g -Wall -o shop item.o shoppingList.o shop.o
item.o : item.cpp item.h
        g++ -g -Wall -c item.cpp
shoppingList.o : shoppingList.cpp shoppingList.h
        g++ -g -Wall -c shoppingList.cpp
shop.o : shop.cpp item.h shoppingList.h
        g++ -g -Wall -c shop.cpp
clean :
        rm -f item.o shoppingList.o
```

Rule



Simplify the Makefile with variables

- Say we want to change
  - 1. The compiler from gcc to clang/icc
  - 2. Change the optimization flags: To -O2 or even remove -g option

```
hello: bar.o foo.o
	gcc -o hello bar.o foo.o

bar.o: bar.c bar.h
	gcc -c -g -O0 bar.c

foo.o: foo.c
	gcc -c -g -O0 foo.c

clean:
	rm -rf hello
	rm -rf bar.o foo.o
```

Simplify the Makefile with variables

- Declare a variable: var_name = value
- Use the value of a variable: $(var_name)
- Similar to bash, only variable types in Makefile are string

```
hello: bar.o foo.o
	gcc -o hello bar.o foo.o

bar.o: bar.c bar.h
	gcc -c -g -O0 bar.c

foo.o: foo.c
	gcc -c -g -O0 foo.c

clean:
	rm -rf hello
	rm -rf bar.o foo.o
```

```
OBJECTS = bar.o foo.o
CFLAGS = -g -O0
CC = gcc
hello: $(OBJECTS)
	$(CC) -o hello $(OBJECTS)

bar.o: bar.c bar.h
	$(CC) -c $(CFLAGS) bar.c

foo.o: foo.c
	$(CC) -c $(CFLAGS) foo.c

clean:
	rm -rf hello
	rm -rf $(OBJECTS)
```

Compile C files to dynamic linked library

- Similar to compiling C files to executable
- Still go through four stages: pre-processing, compilation, assembly and linking
- Except linking stage produced a dynamically linked library instead of an executable file

foo.c

```
#include <stdio.h>
void print_from_foo(void){
  printf("From foo!\n");}
```

bar.c

```
#include <stdio.h>
void print_from_bar (void){
        printf("From bar!\n");}
```

main.c

```
void print_from_foo(void);
void print_from_bar(void);
int main(void)
{
        print_from_foo();
        print_from_bar();
}
```

Makefile examples for generating dll

```
main: main.c libprint.so
        gcc -o main main.c -L`pwd` -lprint
        #-lprint: link main.c with libprint.so, similar to -lpthread
        #-L`pwd`: tells gcc where to find the library
libprint.so foo.o bar.o
        gcc -shared -o libprint.so foo.o bar.o
        #-shared: tell gcc to link foo.o bar.o to a shared library instead of        executables
foo.o: foo.c
        gcc -fPIC -c foo.c
        #-fPIC tells gcc to generate location-independent code
bar.o:bar.c
        gcc -fPIC -c bar.c
        #-fPIC tells gcc to generate location-independent code
```

Homework 7

- Additional references: GCC **Flags**
  - -fPIC: compile directive to output position independent code, a characteristic required by shared library
  - -lXXX: Link with “libXXX.so”
    - Without –L to directly specify the path, /usr/lib is used
  - -L: At compile time, find .so file from this path.
  - -Wl, rpath=.: passes options to the linker. –rpath at runtime finds .so from this path
  - -c: Generate objective code from C code.
  - -shared: Produce a shared code that can be linked with other objects to form an executable

Homework 7

- Divide randall.c into dynamically linked modules and a main program. We don’t want resulting executable to load code that it doesn't need (dynamic loading)
- randall.c = randcpuid.c + randlibhw.c + randlibsw.c + randmain.c
  - **randcpuid.c**: contains code that determines whether the current CPU has the RDRAND instruction. Should include randcpuid.h and include interface described by it.
  - **randlibhw.c**: contains the hardware implementation of the random number generator. Should include randlib.h and implement interface described by it.
  - **randlibsw.c**: contains the software implementation of the random number generator. Should include randlib.h and implement interface described by it.
  - **randmain.c**: contains the main program that glues together everything else. Should include randcpuid.h but not randlib.h. Depending on whether the hardware supports the RDRAND instruction, this main program should dynamically load the hardware oriented or software oriented implementation of randlib.

Homework 7

- 3 steps:
  - Build libraries; load libraries; run functions in libraries.
- Stitch the files together via static and dynamic linking to create the program
- **randmain.c** must use *dynamic loading, dynamic linking* to link up with randlibhw.c and randlibsw.c (using randlib.h)
- Write the **randmain.mk** makefile to do the linking

Homework 7

- randall.c outputs N random bytes of data

  Look at the code and understand it

  - Helper functions that check if hardware random number generator is available, and if it is, generates number
    - Hw RNG exists if RDRAND instruction exists
    - Uses cpuid to check whether CPU supports RDRAND (30th bit of ECX register is set)
  - Helper functions to generate random numbers using software implementation (/dev/urandom)
  - Main function
    - Checks number of arguments (name of program, N)
    - Converts N to long integer, prints error message otherwise
    - Uses helper functions to generate random number using hw/sw