CS 35L Software Construction Lab
Week 8 – Dynamic Linking

Anatomy of Linux shared libraries

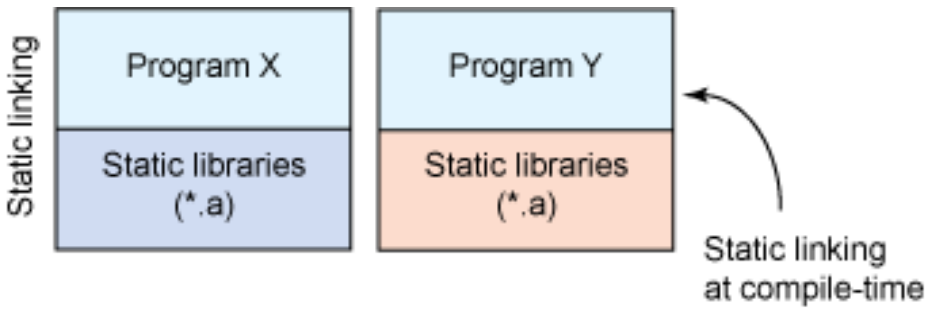
- Libraries - to package similar functionality → modular programming
- Linux supports two types

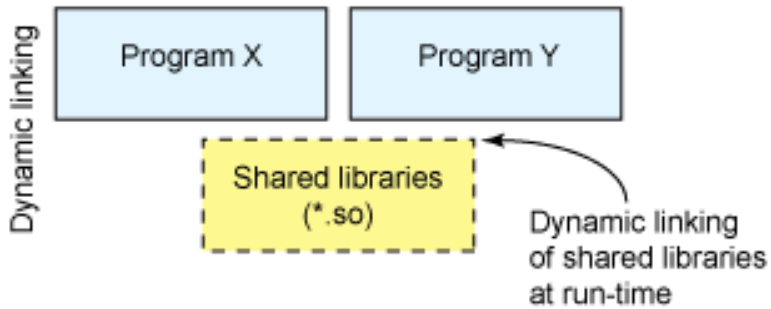
static library

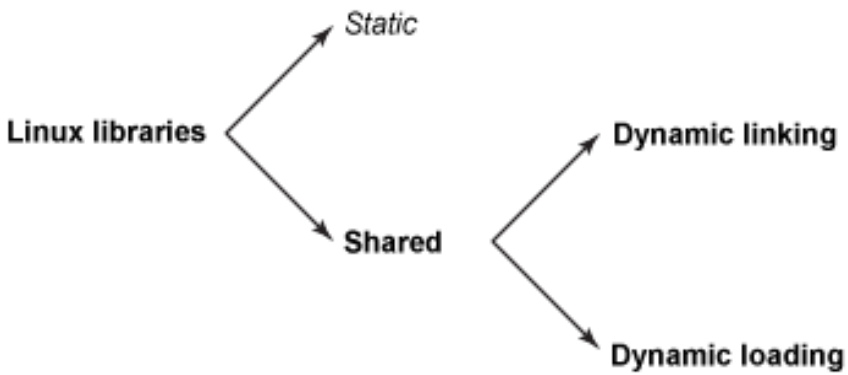
functionality to bind to a program statically at compile-time

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

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

Dynamic Loading

to 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

done by a library named ld-linux.so.2

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

```
#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;
}
```

Creating 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 -cvq 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

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");
}
```

Homework 8

the homework - to split an application into dynamically linked modules
randall.c = randcpuid.c + randlibhw.c + randlibsw.c + randmain.c

randall.c =
randcpuid.c + randlibhw.c + randlibsw.c + randmain.c

1. build the libraries
2. load the libraries
3. run the functions in libraries

Homework 8

Flags:

gcc -shared -fPIC greeting-fr.c -o greeting-fr.so

gcc -ldl -Wl,-rpath=. greeting-dl.c -o greet-dl

- -fPIC to output position independent code
- -lmylib to link with \libmylib.so"
- -L to nd .so les from this path, default is /usr/lib
- -Wl,rpath=dir to set rpath option to be dir to linker (by using -Wl)
- -shared to build a shared object

Attribute of functions:

__attribute__ ((constructor)) to run when dlopen() is called

__attribute__ ((destructor)) to run when dlclose() is called

Homework 8

- 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)
  - **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 8

- 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 8

- 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