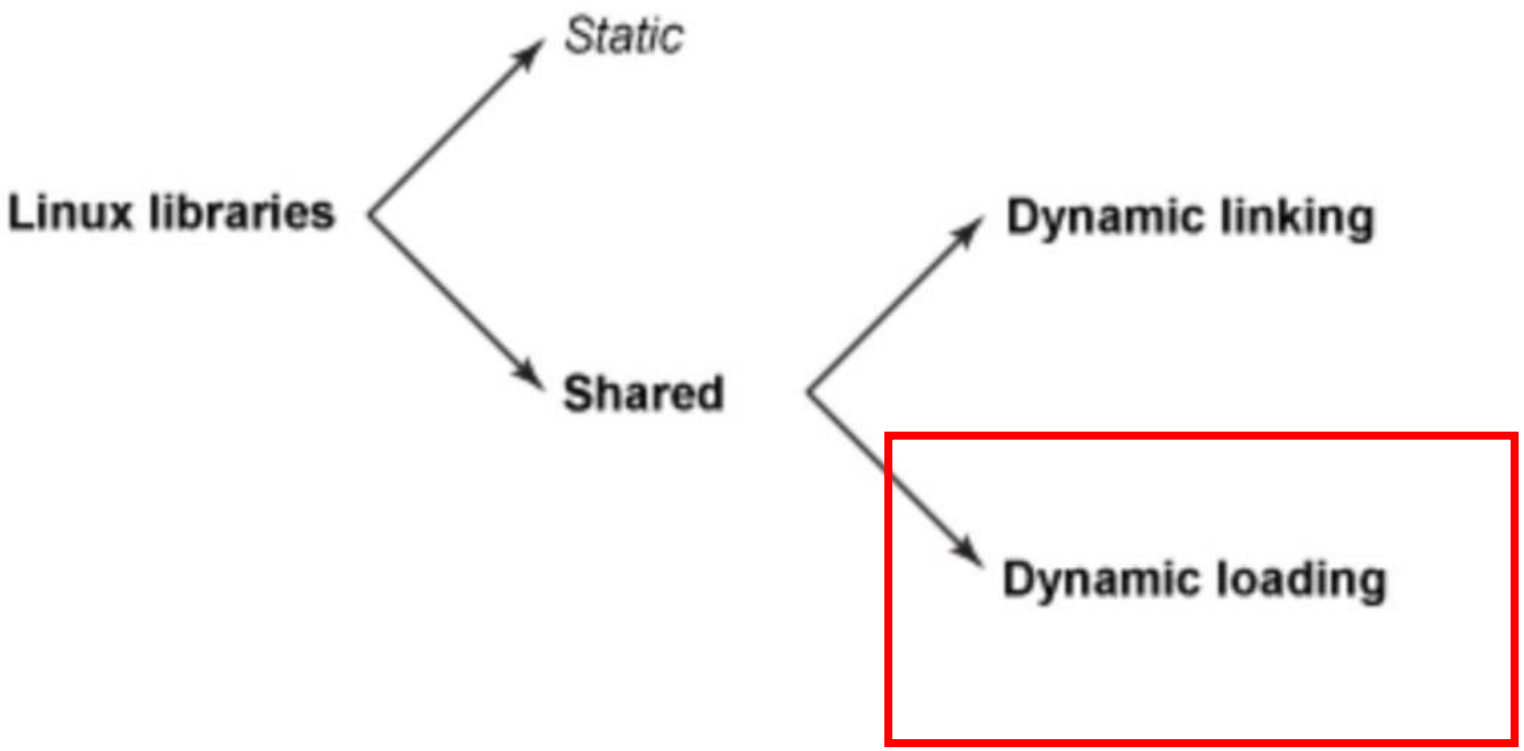
UCLA CS35L

Week 6

Thursday

Library Hierarchy in Linux



Dynamic Loading

- With dynamic linking, we had the OS help us to automatically link and load libraries at start time.
- With dynamic loading, we/the application will manually specify libraries to load during runtime of the application
- We use the DL API to accomplish this
  - Include <dlfcn.h>

- NOTE – we do NOT need to #include the actual library file now. We just open it directly in the body of our code

DL API - Overview

- dlopen – Makes an object file accessible to the program
- dlsym – Obtains the resolved address of a symbol within an opened object
- dlerror – Returns a string error of the last error that occurred
- dlclose – Closes an object file

DL API - dlopen

```
void *dlopen(const char* file, int mode);
```

- Input
  - const char* file – A c-string to the .so file
  - mode – specifies when relocation will be done
    - RTLD_NOW – complete at dlopen call time
    - RTLD_LAZY – complete when needed
- Output
  - A handle to the object library

DL API - dlsym

```
void* dlsym(void* restrict handle, const char* restrict name);
```

- Input
  - void* handle – The output of dlopen
  - char* name – The symbol name defined in the object
- Output
  - Resolved address of that symbol

DL API - dlerror

```
char* dlerror();
```

- Input
  - None
- Output
  - C-string of the last error that occurred
  - Error will be in human-readable form

DL API - dlclose

```
int dlclose(void* handle);
```

- Input
  - void* handle – the output of dlopen
- Output
  - Return 0 if was able to successfully call dlclose
  - Otherwise return non-zero

- NOTE – This function dereferences the handle object.
  - Only once there are no more references to that object, does handle get removed from memory.

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

int main(){
    //open .so handle, can also use RTLD_NOW
    void *dl_handle;
    dl_handle = dlopen("libAdd1.so", RTLD_LAZY);
    if (!dl_handle){
        fprintf(stderr, "dlopen() error - %s\n", dlerror());
        return 1;
    }

    //load add1 function
    int (*myAdd1Func)(int);
    myAdd1Func = dlsym(dl_handle, "add1");
    char *error = dlerror();
    if (error != NULL){
        fprintf(stderr, "dlsym() error with add1 - %s\n", error);
        return 1;
    }

    //use function and close handle
    int x = 1;
    int y = myAdd1Func(x);
    printf("Before - %d, After - %d\n", x, y);
    dlclose(dl_handle);
    return 0;
}
```

DL Compiling

```
#Compile with fpic to output "position independent code"
gcc -fPIC -c add1.c

#Compile as shared library, and pass name to linker
gcc -shared -Wl,-soname,libAdd1.so.1 -o libAdd1.so.1.0 add1.o

#Create symbolic links to original file
ln -sf libAdd1.so.1.0 libAdd1.so.1
ln -sf libAdd1.so.1 libAdd1.so

#Compile loadAdd, and still set rpath to current dir '.'
# -ldl flag tells exe to use to dynamic loading
gcc loadAdd1.c -o loadAdd -ldl -Wl,-rpath,.
```

- NOTE – Only changes are all in compiling the .exe
  - Still need to create shared library with same process as before
- Exe compilation needs the –ldl flag and the –Wl,-rpath,. setting

__attribute__

- You can add to your libraries to add more behaviors
- __attribute__ ((__constructor__))
  - runs whatever is in here when dlopen() is called
- __attribute__((__destructor__))
  - runs whatever is in here when dlclose() is called

attribute example

```c
#include <stdio.h>

void __attribute__ ((__constructor__)) printEnter();
void __attribute__ ((__constructor__)) printExit();

void printEnter(){
    printf("Entering library - attribute constructor call\n");
}
void printExit(){
    printf("Exiting library - attribute destructor call\n");
}

int add1(int n){
    return n+1;
}
```

Makefile

- Makefile is a build tool that we have been using, and was introduced in week 3
- Designed and written in a few parts:
  - Targets have dependencies
    - If the dependency has not been built yet, go find out how to build that dependency (should be a target somewhere else)
  - Once all the dependencies for a target are ready, execute the given commands
    - Commands are executed in their own subshell. So can be almost any Bash command, not just Compiler commands

Makefile Example – For homework

```
OPTIMIZE = -O2

CC = gcc
CFLAGS = $(OPTIMIZE) -g3 -Wall -Wextra -march=native -mtune=native -mrdrnd

default: libAdd mainAdd

#include instructions for building libAdd and mainAdd
-include add.mk

clean:
    rm -f *.a *.o *.so *.so.* makeAdd1
```

- You will be given an overall Makefile like the above. It will do the following:
  - Specify some variables for you to use (CC and CFLAGS)
  - Some targets for the make commands
- Includes a sub-file (.mk file) which you will write targets/commands for

What do you need to add?

- You need to convert your compiler commands, into a Makefile form
- Below has the same compiler commands we reviewed in the dynamic loading case, but now formatted for a Makefile
  - Using the CC and CFLAGS variables for Compiler details
  - Specify targets, and their dependencies
  - And once all dependencies are met, run the compiler commands like before

```
libAdd: add1.c add1.h
    $(CC) $(CFLAGS) -fPIC -c add1.c -o add1.o
    $(CC) $(CFLAGS) add1.o -o libAdd1.so.1.0 -shared -Wl,-soname,libAdd1.so.1
    ln -sf libAdd1.so.1.0 libAdd1.so.1
    ln -sf libAdd1.so.1 libAdd1.so

mainAdd: libAdd loadAdd1.c
    $(CC) $(CFLAGS) loadAdd1.c -o makeAdd1 -ldl -Wl,-rpath=$(PWD)
```

My tips for Makefiles

- Figure out how many independent parts of the code you can compile
- Try to compile each piece individually, by manually typing commands
- Once you figure out what is required for that piece, convert to:
  - Target: Dependencies
  - Commands to complete the target

- Build all those pieces together to create the entire program executable

ldd command

- Used to see what dynamic libraries are used by a program

```
lnxsrv09:~/CS_Homework/CS35L/week6_linking$LD_LIBRARY_PATH="." ldd ./add1_dynamic
        linux-vdso.so.1 =>  (0x00007fff7e7fb000)
        libAdd1.so.1 => ./libAdd1.so.1 (0x00007fce7cfcf000)
        libc.so.6 => /lib64/libc.so.6 (0x00007fce7cc01000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fce7d1d1000)
```

Library Interpositioning

- We replace calls to library functions with calls to our custom made wrapper.
- Can be done at
  - Compile Time
  - Link Time
  - Run Time
- Example – replace call to malloc with our own custom malloc which prints debug information
  - https://www.geeksforgeeks.org/function-interposition-in-c-with-an-example-of-user-defined-malloc/

Lab 6 Hints

- Will need to use ldd and strace, may also find it helpful to use head, tail, and wc
- If you want to pass 2^24 as a parameter, use `./run $((2**24))`
- Recommend a shell script for the last part (run ldd on 9 commands) Something like below, and then can pipe output to grep and sort.

```
#!/bin/bash
for x in "$(ls /usr/bin | awk '(NR-your_uid)%251 == 0')"; do
    y=`which $x`
    ldd $y
done
```