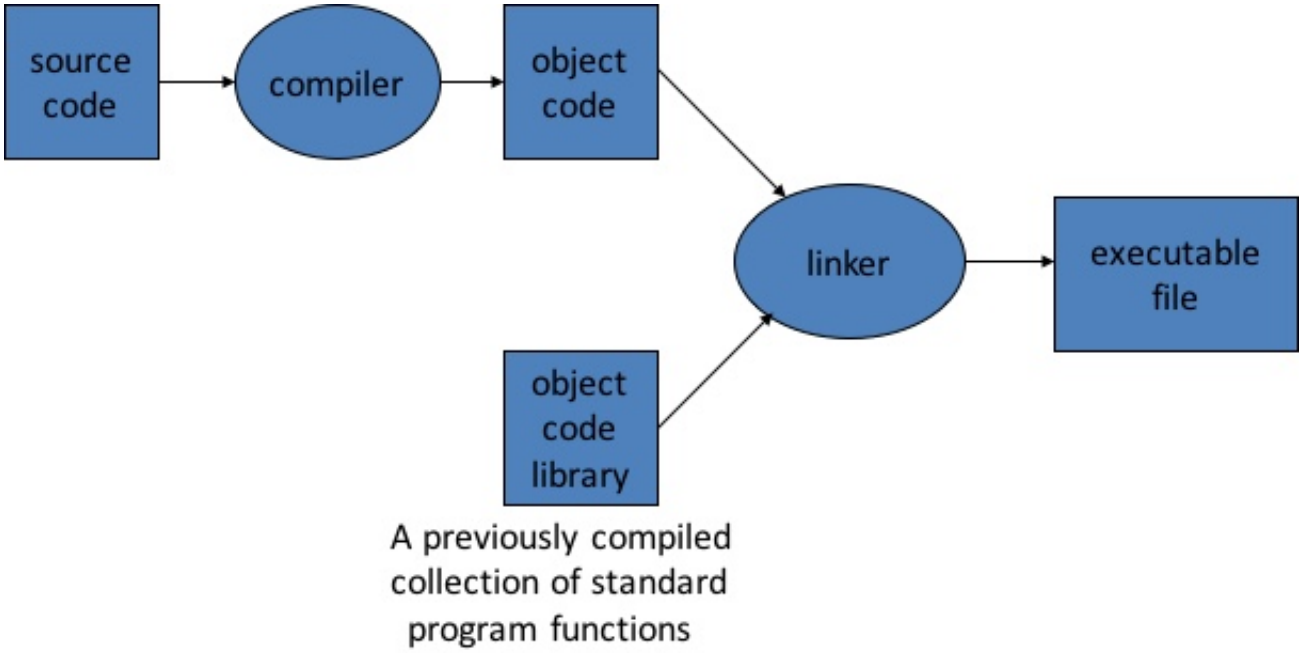
Week 9: Dynamic Linking

Building an Executable File

- translates programming language statements into CPU's machine-language instructions
- adjusts any memory references to fit the OS's memory model

Static Linking



- carried out only once to produce an executable file
- if static libraries are called, the linker will copy all the modules referenced by the program to the executable
- static libraries are typically denoted by the `.a` file extension

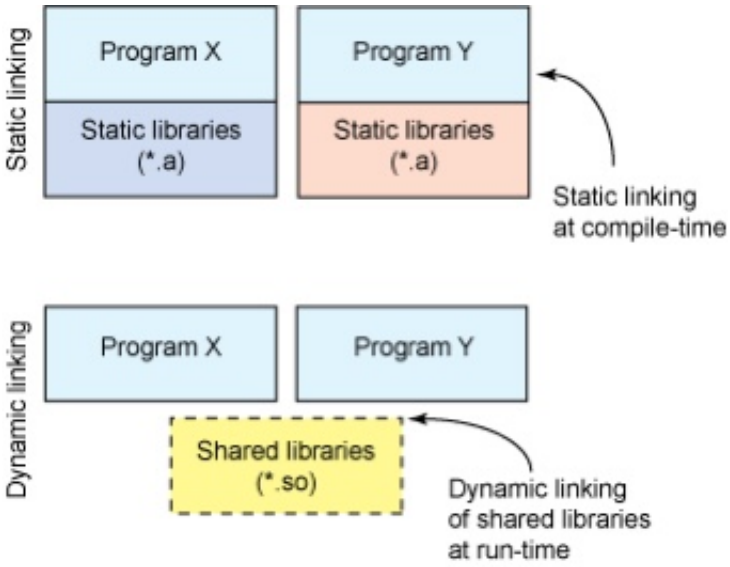
Dynamic Linking

- allows a process to add, remove, replace or relocate object modules during its 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
        - loading time is when process is first called
        - running time is at some point while process is running
- dynamic libraries are typically denoted by the `.so` file extension (shared objects)

- `.dll` on Windows (dynamically linked library)

Linking and Loading

- **loading**: disk ➤ RAM
- **linking**: resolving addresses from memory space of libraries to that of the process itself
- linker collects procedures and links them together object modules into one executable program
- *Why isn't everything written as just one **big** program, saving the necessity of linking?*
  - **efficiency**: if just one function is changed in a 100K line program, why recompile the whole program; just recompile one function and relink
  - multiple-language programs



- Unix systems: code is typically compiled as a *dynamic shared object* (DSO)

```
# Dynamic vs. static linking resulting size
$ gcc -static hello.c -o hello-static   # static linking
$ gcc hello.c -o hello-dynamic          # dynamic linking
$ ls -l hello
    80 hello.c
 13724 hello-dynamic
   383 hello.s
1688756 hello-static                    # HUGE filesize!
```

How Libraries are Dynamically Loaded

Table 1. The 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

dlopen

- makes a library accessible to a program by loading it to RAM from disk

dlsym

- resolves addresses of linked libraries

dlerror

- returns error if something went wrong

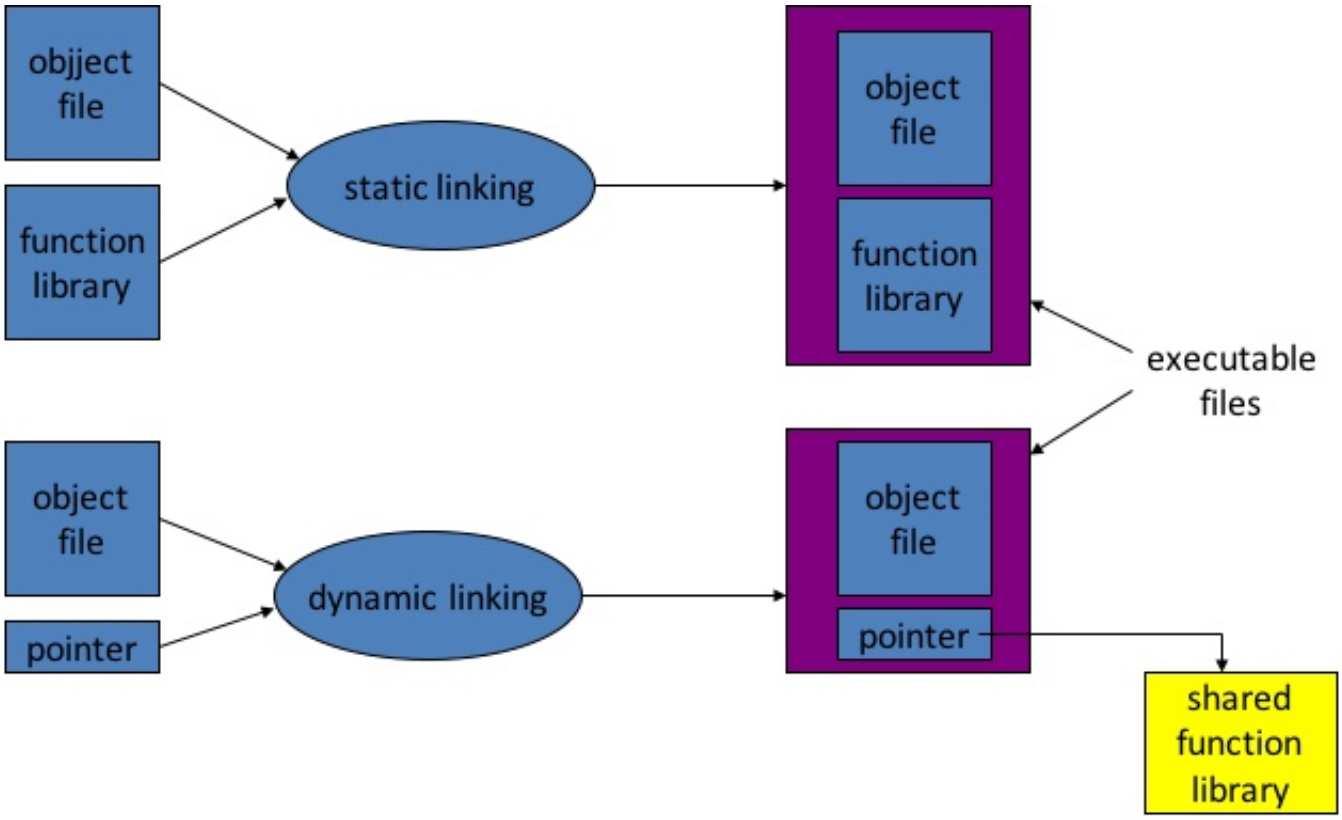
dlclose

- unloads a dynamic library from RAM (if no other processes are using it)

Advantages of Dynamic Linking

- executable is typically smaller
- when the library is changed, the code that references it does not usually need to be recompiled
- the executable accesses the `.so` at runtime
- therefore, multiple programs can access the same `.so` at the same time
    - memory footprint amortized across all programs using same `.so`

Smaller is more efficient



Disadvantages of Dynamic Linking

- performance hit
  - need to load shared objects (at least once)
  - need to resolve addresses (once or every time)
  - remember back to the system call assignment…
- *What if the necessary dynamic library is missing?*
- *What if we have the library, but it is the wrong version?*

Lab 9

- write and build a simple “cos(0.5)” program in C
  - use `ldd` to invesigate which dynamic libraries your cos program loads
  - use `strace` to investigate which system calls your cos program makes
- use `ls /usr/bin | awk 'NR%101==SID%101'` to find ~25 linux commands to use ldd on
  - record output for each one in your log and investigate any errors you might see
  - from all dynamic libraries you find, create a sorted lsit
    - remember to remove the duplicates!

GCC Flags

- `fPIC`: compiler directive to output position independent code, a characteristic required by shared libraries
- `-lXXX`: link with "`libXXX.so`"
    - without `-L` to directly specify the path, /usr/lib is used
- `-L`: at compile time, find .so from this path
- `-Wl,rpath=.`: `-Wl` passes options to linker
    - `-rpath` at runtime finds .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
- https://gcc.gnu.org/onlinedocs/gcc/Link-Options.html#Link-Options

Creating static and shared libs in GCC

```c
// 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)
```

Dynamic Loading

```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 will have to set the environment variable `LD_LIBRARY_PATH` to include the path that contains libmymath.so

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

```
__attribute__ ((__constructor__))
// is run when dlopen() is called

__attribute__ ((__destructor__))
// is run when dlclose() is called

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

Homework 9

- split **`randall.c`** into 4 separate files
- 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

- `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
      - 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 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

---

- divide **`randall.c`** into dynamically linked modules and a main program
  - don't want resulting executable to load code that it doesn't need (dynamic loading)
    - **`randcpuid.c`**: contains code that determines whether the CPU has the RDRAND instruction; should include randcpuid.h and include interface described by it
    - **`randlibhw.c`**: contains hardware implementation of the random number generator; should include randlib.h and implement the interface described by it
    - **`randlibsw.c`**: contains the software implementation of the random number generator; shoul dinclude randlib.h and implement the interface described by it
    - **`randmain.c`**: contains the main program that glues together everything else; should include randcpuid.h (as the corresponding module should be linked statically) but not randlib.h (as the corresponding module should be linked after main starts up). Depending on whether the hardware supports the RDRAND instruction, this main program should dynamically load the hardware-oriented or software-oriented implementation of randlib.