

# CS 35L

Week 9

TA: Tomer Weiss  
March-01-2016

goo.gl/Qmc8Yj

Slides

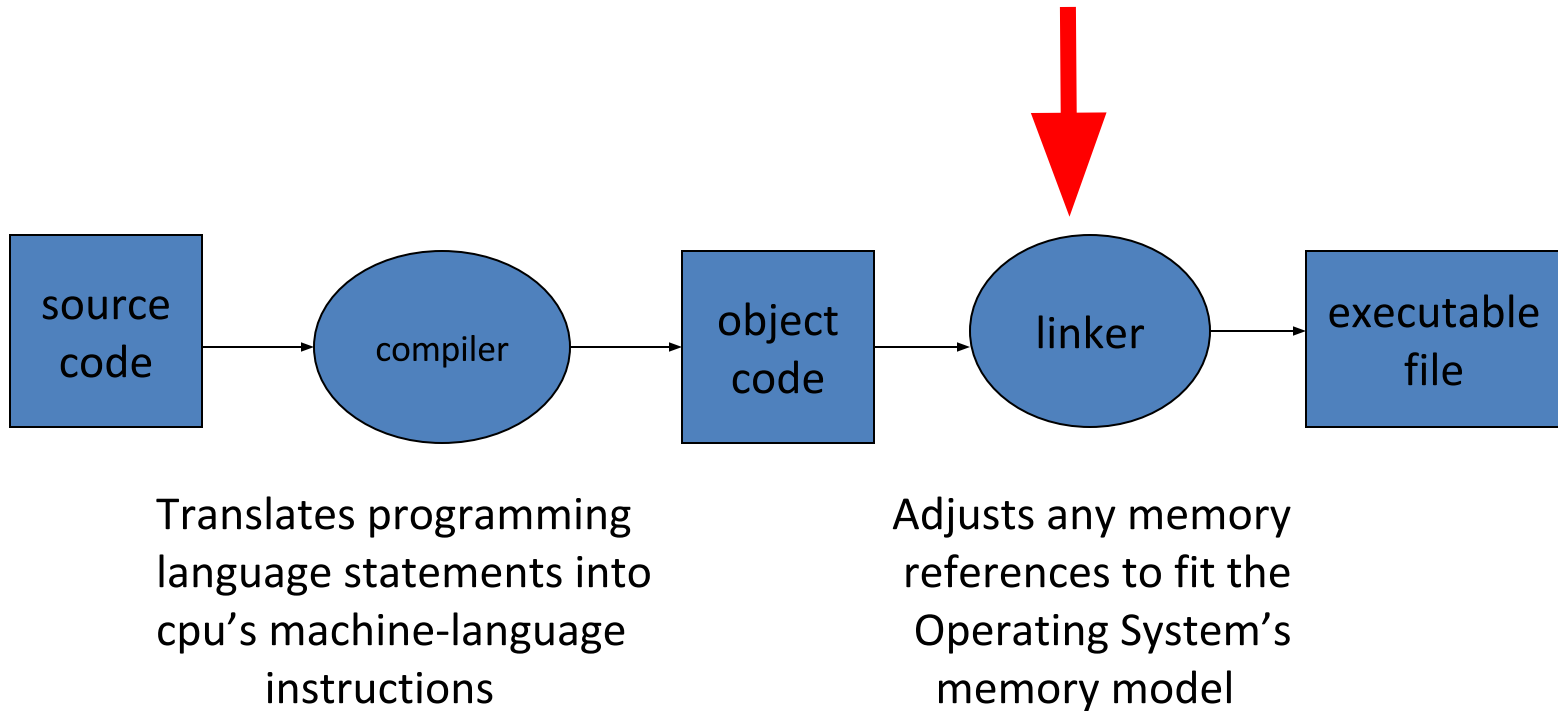
# Announcements

- Student presentations today:
  - CSI: Cyberattack Scene Investigation--a Malware Whodunit by Zhuoran
  - Extracting Translation Pairs from Social Network Content by Weifeng
  - Disney's VertiGo Combines Car, Helicopter to Drive Up Walls by Yuxing
- [web.cs.ucla.edu/classes/winter16/cs35L/assign/assign10.html](http://web.cs.ucla.edu/classes/winter16/cs35L/assign/assign10.html)
- Next week:
  - Write your topic [here](#)
  - Not registering your topic beforehand may result in rescheduling of your presentation
  - For reference on presentation, grading, please refer to this [rubric](#).

# Dynamic Linking

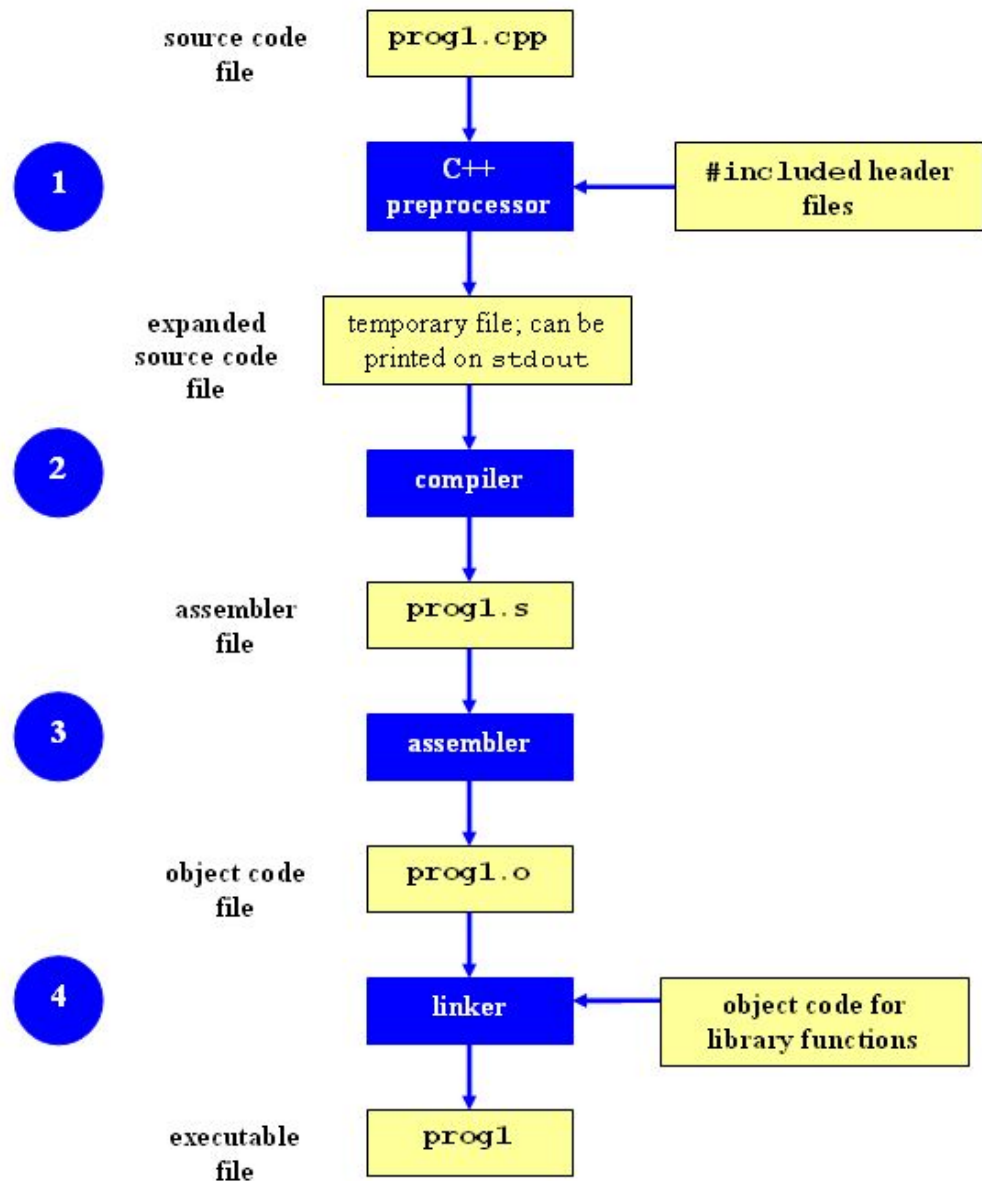
Week 9

# Building an executable file



# Compilation Process

- Preprocessor
  - Expand Header includes, macros, etc
  - -E option in gcc to show the resulting code
- Compiler
  - Generates machine code for certain architecture
- Linker
  - Link all modules together
  - Address resolution
- Loader
  - Loads the executable to memory to start execution



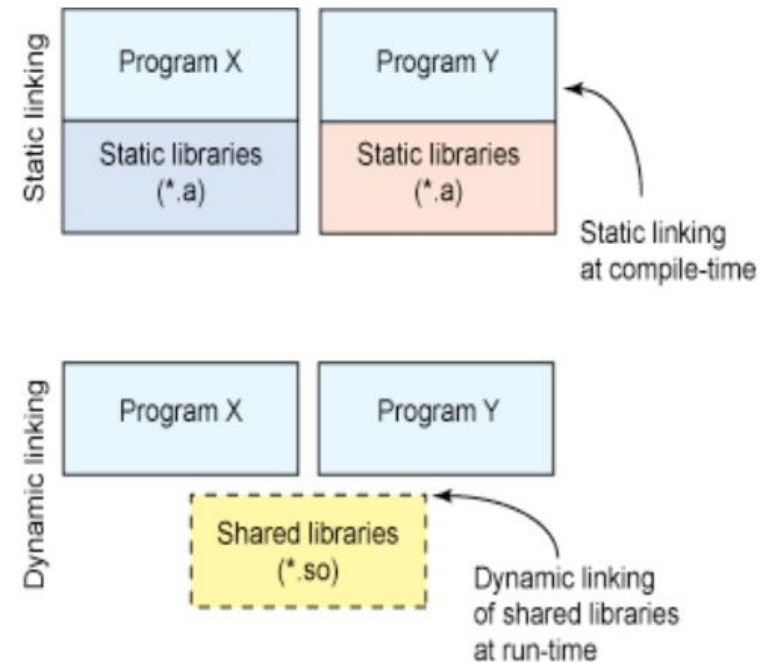
# Linux Libraries

- **Static Library**

- Statically linked
- Every program has its own copy
- More space in memory
- Tied to a specific version of the lib. New version of the lib requires recompile of source code.

- **Shared Library (binding at run-time)**

- Dynamically loaded/linking
  - **Dynamic Linking** – The OS loads the library when needed. A dynamic linker does the linking for the symbol used.
  - **Dynamic Loading** – The program “actively” loads the library it needs (DL API – dlopen(), dlclose()). More control to the program at run-time. Permits extension of programs to have new functionality.
- Library is shared by multiple programs
- Lower memory footprint
- New version of the lib does not require a recompile of source code using the lib

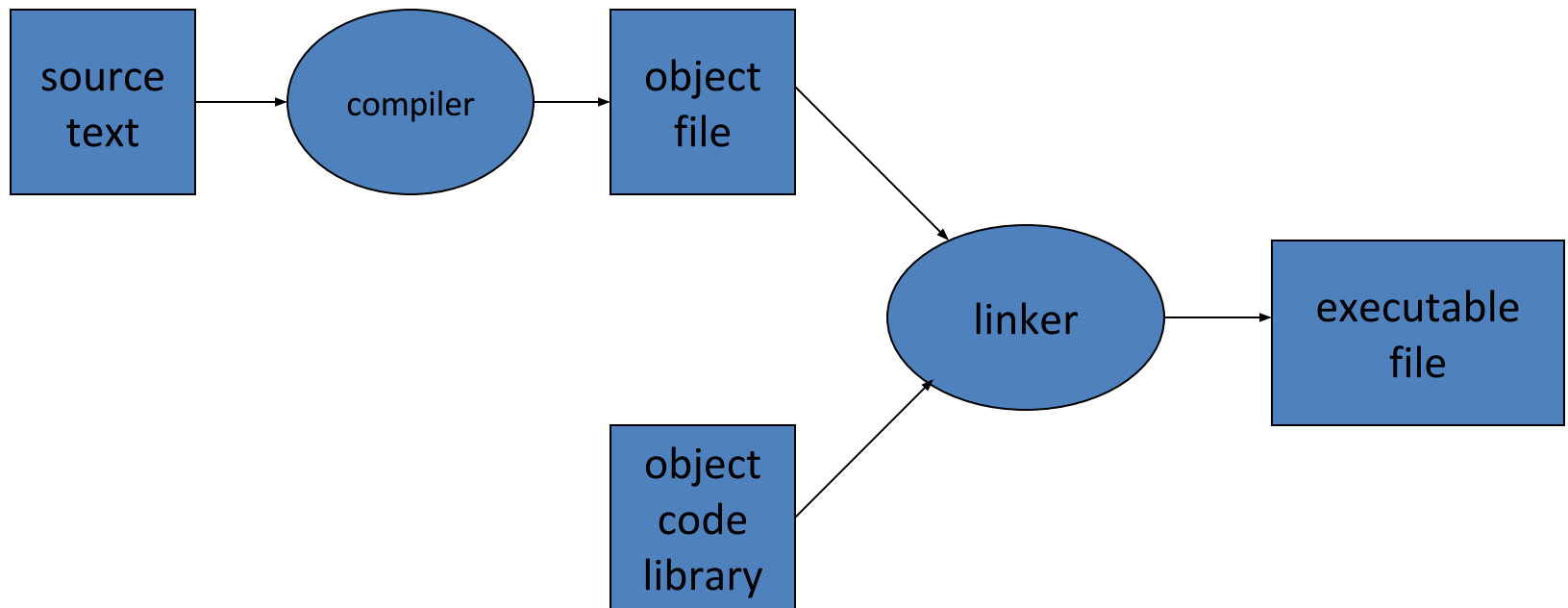


Img Source : <http://www.ibm.com/developerworks/library/l-dynamic-libraries/>

# 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





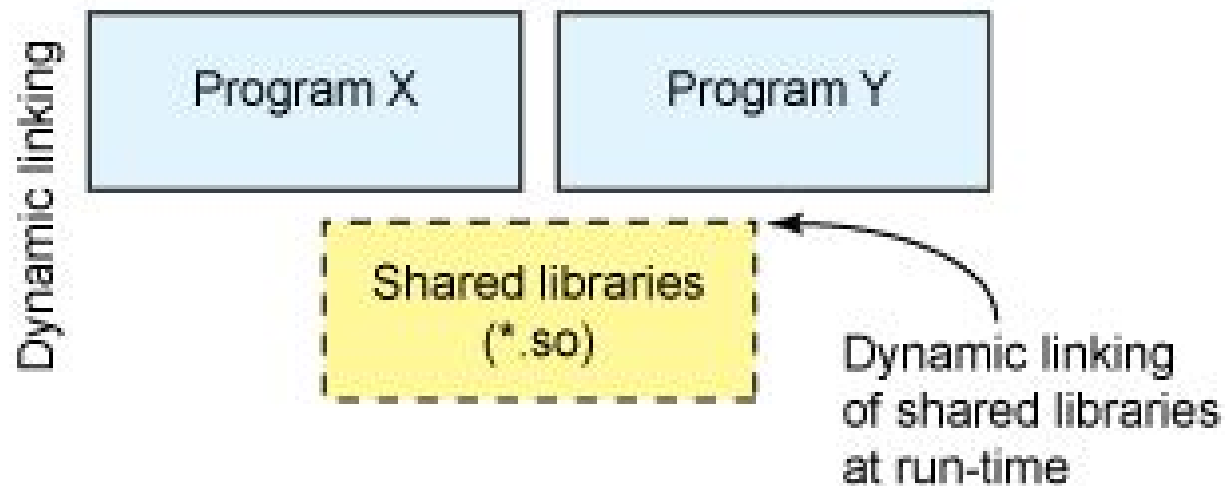
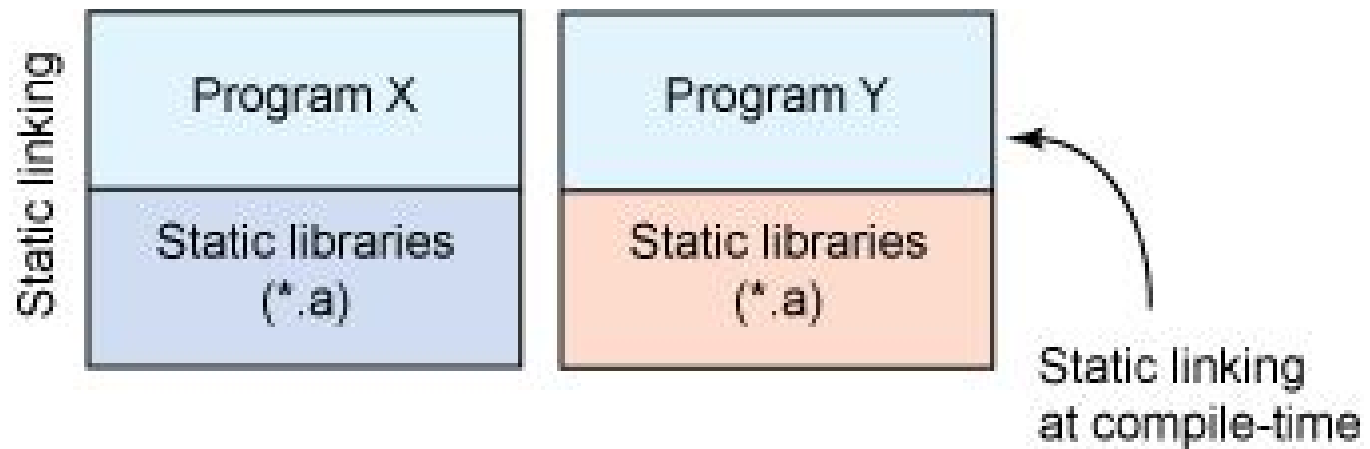
A previously compiled  
collection of standard  
program functions

# 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
- Dynamic libraries are typically denoted by the .so file extension
  - .dll on Windows

# Linking and Loading

- 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 the one function and relink.
  - Multiple-language programs
  - Other reasons?



# Dynamic linking

- 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
```

```
$ gcc hello.c -o hello-dynamic
```

```
$ ls -l hello
```

```
    80 hello.c
```

```
 13724 hello-dynamic
```

```
   383 hello.s
```

```
1688756 hello-static
```

- If you are the sysadmin, which do you prefer?

# How are libraries dynamically linked?

**Table 1. The DL API**

Function	Description
<b>dlopen</b>	Makes an object file accessible to a program
<b>dlsym</b>	Obtains the address of a symbol within a dlopened object file
<b>dlerror</b>	Returns a string error of the last error that occurred
<b>dlclose</b>	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;
}
```

- 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

# GCC options

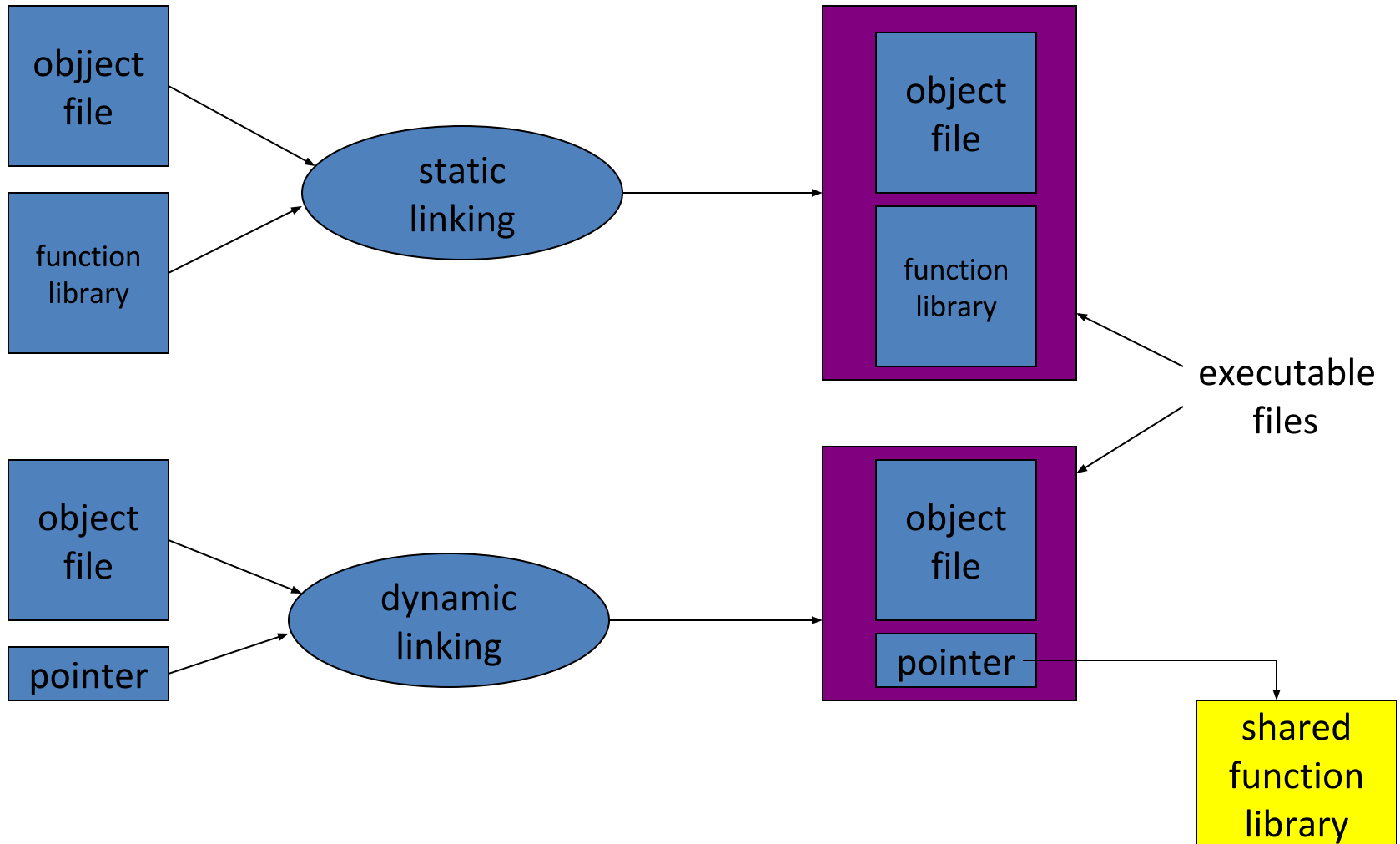
- -c : compile and create object files
- -o : name of output file
- -I (upper case i) : Additional folders to search for header files
- -L : Additional folders to search for libraries to link with
- -shared : create shared library
- -l (lower case 'ell') : Name of additional library to link with
- -fpic : Output position independent code. This is required for shared libraries (generating relative addresses instead of absolute addresses)



# Advantages of dynamic linking

- The 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 run time; therefore, multiple programs can access the same .so at the same time
  - Memory footprint amortized across all programs using the 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 simple math program in C
  - compute `cos(0.5)` and print it using the `printf` format `"%.17g"`
  - Use `ldd` to investigate which dynamic libraries your hello world program loads
  - Use `strace` to investigate which system calls your hello world program makes
- Use `"ls /usr/bin | awk 'NR%101==SID%101'"` to find +-23 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 list
    - Remember to remove the duplicates!

## *Lab 9 hint*

```
#!/bin/bash
```

```
for x in "$(ls /usr/bin | awk 'NR%101==your_uid%101'
$1)"; do
    y=`which $x`
    ldd $y
done
```

example run, unique sort, need to omit addresses at end:

```
./ldd_run | grep so | sort -u
```

# Pointers

- Useful overview of dynamic libraries: [Click here](#)
- Man page for DL API: [Click here](#)
- [www.ibm.com/developerworks/library/l-dynamic-libraries/](http://www.ibm.com/developerworks/library/l-dynamic-libraries/)
- [www.ibm.com/developerworks/library/l-lpic1-102-3/](http://www.ibm.com/developerworks/library/l-lpic1-102-3/)
- [tldp.org/HOWTO/Program-Library-HOWTO/index.html](http://tldp.org/HOWTO/Program-Library-HOWTO/index.html)
- [www.yolinux.com/TUTORIALS/LibraryArchives-StaticAndDynamic.html](http://www.yolinux.com/TUTORIALS/LibraryArchives-StaticAndDynamic.html)
- [man7.org/linux/man-pages/man7/vdso.7.html](http://man7.org/linux/man-pages/man7/vdso.7.html)

# Lab

[web.cs.ucla.edu/classes/winter16/cs35L/assign/assign9.html](http://web.cs.ucla.edu/classes/winter16/cs35L/assign/assign9.html)

# 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.



# 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)

# 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;
}
```

- 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

# 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 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

# Homework 9

- 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 (30<sup>th</sup> 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

# Homework 9

- 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 implement interface described by it.
    - **randlibhw.c**: contains the 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. Should include 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.

# Homework

[web.cs.ucla.edu/classes/winter16/cs35L/assign/assign9.html](http://web.cs.ucla.edu/classes/winter16/cs35L/assign/assign9.html)