Lecture # 1.1
C-Compilation & Libraries

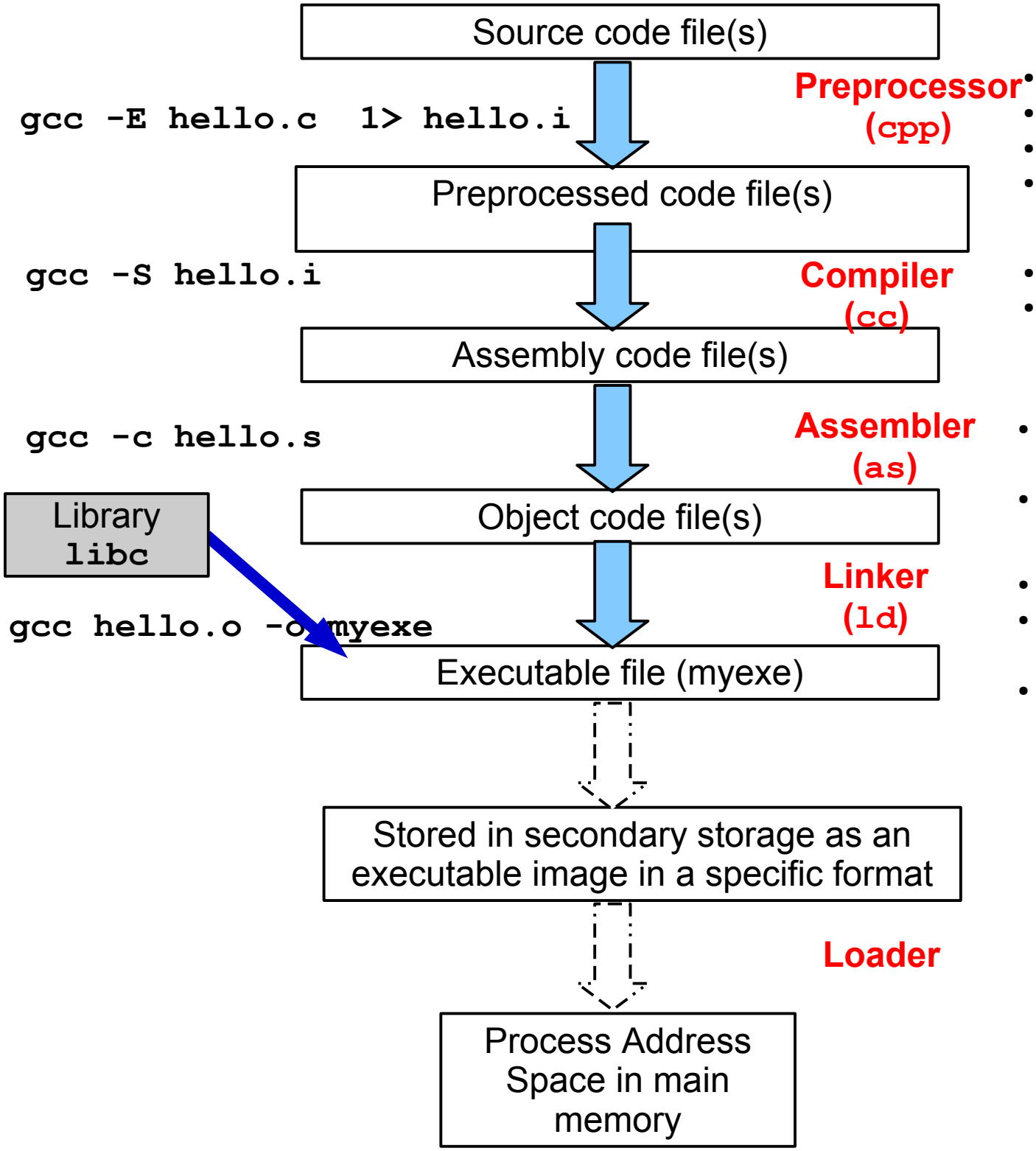
Course: Advanced Operating System

Instructor: Arif Butt

Punjab University College of Information Technology (PUCIT)
University of the Punjab

Today's Agenda

- Review of C Compilation process
- Format of object files
- Viewing the contents of object files
- Loading a program in memory
- Layout of a process in memory
- Review of Linking process
- Merging Relocatable Object Files into Executable
- Understanding Linker relocation process
- Understanding Linker symbol resolution
- Creating and using your own Static Libraries
- Creating and using your own Dynamic Libraries



- Interpret preprocessor directives
- Include header files
- Expand macros
- Remove comments

- Checks for syntax errors
- Converts the src to assembly of underlying processor

- Generates relocatable object files to be used by linker
- Contains symbol table

- Static vs Dynamic linking
- Contains code and data for all functions defined in src files
- Contains global symbol table

Types of Object Files (Modules)

- **Relocatable object file: (.o file)** Contains binary code and data in a form that can be combined with other relocatable object files at compile time to create an executable object file. Each .o file is produced from exactly one .c file. Compilers and assemblers generate relocatable object files.
- **Executable object file: (a.out file)** Contains binary code and data in a form that can be copied directly into memory and executed. Linkers generates executable object files.
- **Shared object file: (.so file)** A special type of relocatable object file that can be loaded into memory and linked dynamically, at either load time or run time. Called dynamic link libraries (dlls) in Windows. Compilers and assemblers generate shared object files.
- **Core file:** A disk file that contains the memory image of the process at the time of its termination. This is generated by system in case of abnormal process termination.

Formats of Object Files (Modules)

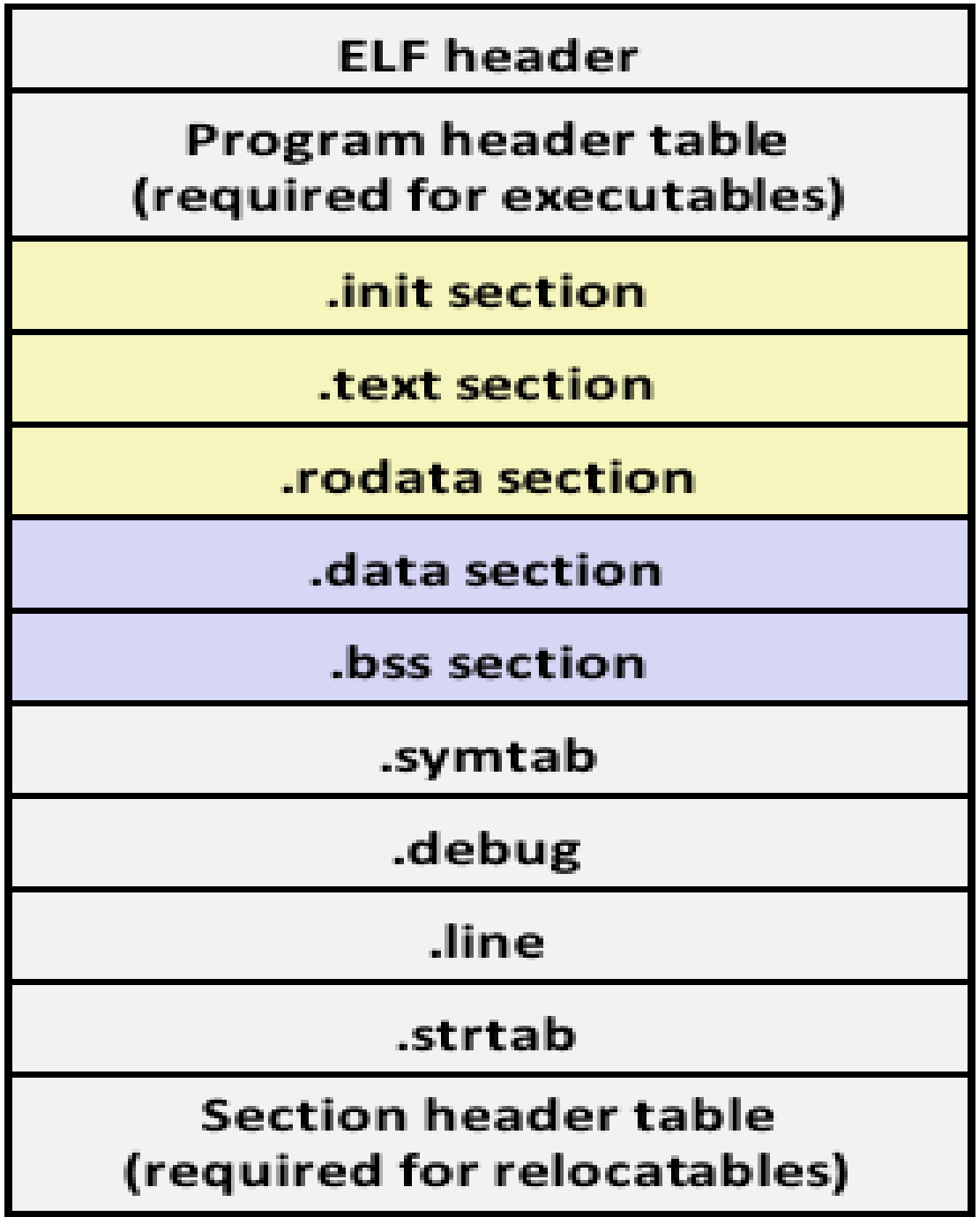
Object file formats vary from system to system. Some famous formats are mentioned below:

Formats	Description
a.out	Original file format for UNIX. It consists of **three** sections: text, data, and bss, which are for program code, initialized data and uninitialized data respectively.
COFF	Common Object File Format was introduced with SVR3 Unix. COFF files may have multiple sections, each prefixed by a header. The number of sections is limited. The COFF specification includes support for debugging but the debugging info was limited. Later ECOFF was introduced by MIPS and XCOFF by IBM
ELF	Executable and Linking Format came with SVR4 UNIX. ELF is similar to COFF in being organized into a number of sections, but it removes many of COFF's limitations. ELF is used on most modern UNIX systems, including GNU/Linux, Solaris and Irix. Also used on many embedded systems
PE	Portable Executable format is used by Windows for their executables. PE is basically COFF with additional headers. The extension normally is .exe

ELF Format

- Executable and Linking Format is binary format, which is used in SVR4 Unix and Linux systems
- It is a format for storing programs or fragments of programs on disk, created as a result of compiling and linking
- ELF not only simplifies the task of making shared libraries, but also enhances dynamic loading of modules at run time
- An executable file using the ELF format consist of ELF Header, Program Header Table and Section Header Table
- The files that are represented in this formats are:
  - Relocatable file objects (**.o**)
  - Normal executable files (**a.out**)
  - Shared object files (**.so**)
  - Core files

ELF Format (cont...)



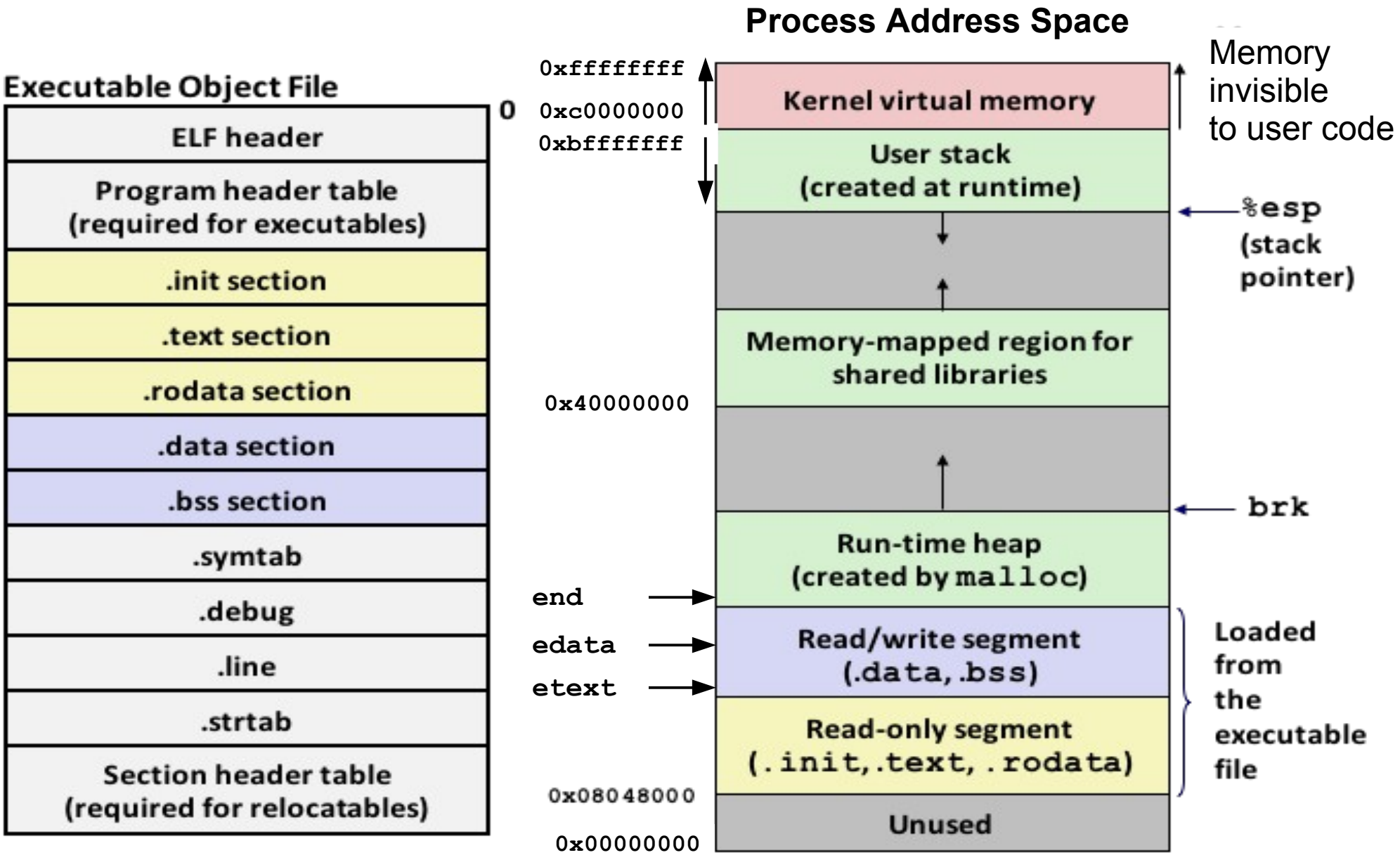
Reading Contents of Object Files

`readelf, objdump`

Loading program in gdb
&
Viewing CPU Registers

Loading executable in Memory

Loading Executable File in Memory



Startup Routine in `crt1.o`

```
0x08048000 <_start>:  /* entry point in .text  */
call __libc_init_first/* startup code in .text */
call _init            /* startup code in .init */
call atexit            /* startup code in .text */
call main             /* application main func */
call _exit            /* returns control to OS*/
```

Linking Process

Why should you Learn Linking Process?

- Understanding linkers will help you build large programs
- Understanding linker will help you avoid dangerous programming errors, like what happens when you create global variables with same name in multiple object files?
- Understating linking will help you understand how language scoping rules are implemented, like what happen when you declare a variable or function with static attribute?
- Understanding linking will help you understand other system concepts like loading and running programs, virtual memory, paging, and memory mappings
- Understanding linking will enable you to exploit shared libraries

Why should you Learn Linking Process?

Advantages of Linkers

- **Modularity:** Programs can be written as a collection of smaller source files rather than one monolithic mass. We can build libraries of common function like the standard C library `/usr/lib/x86_64-linux-gnu/libc.a`
- **Efficiency:** It saves time, e.g., if we have ten source files and have made change in only one, we need to compile only that file and not all the files. Later of course we need to relink all the object files

What Linkers do?

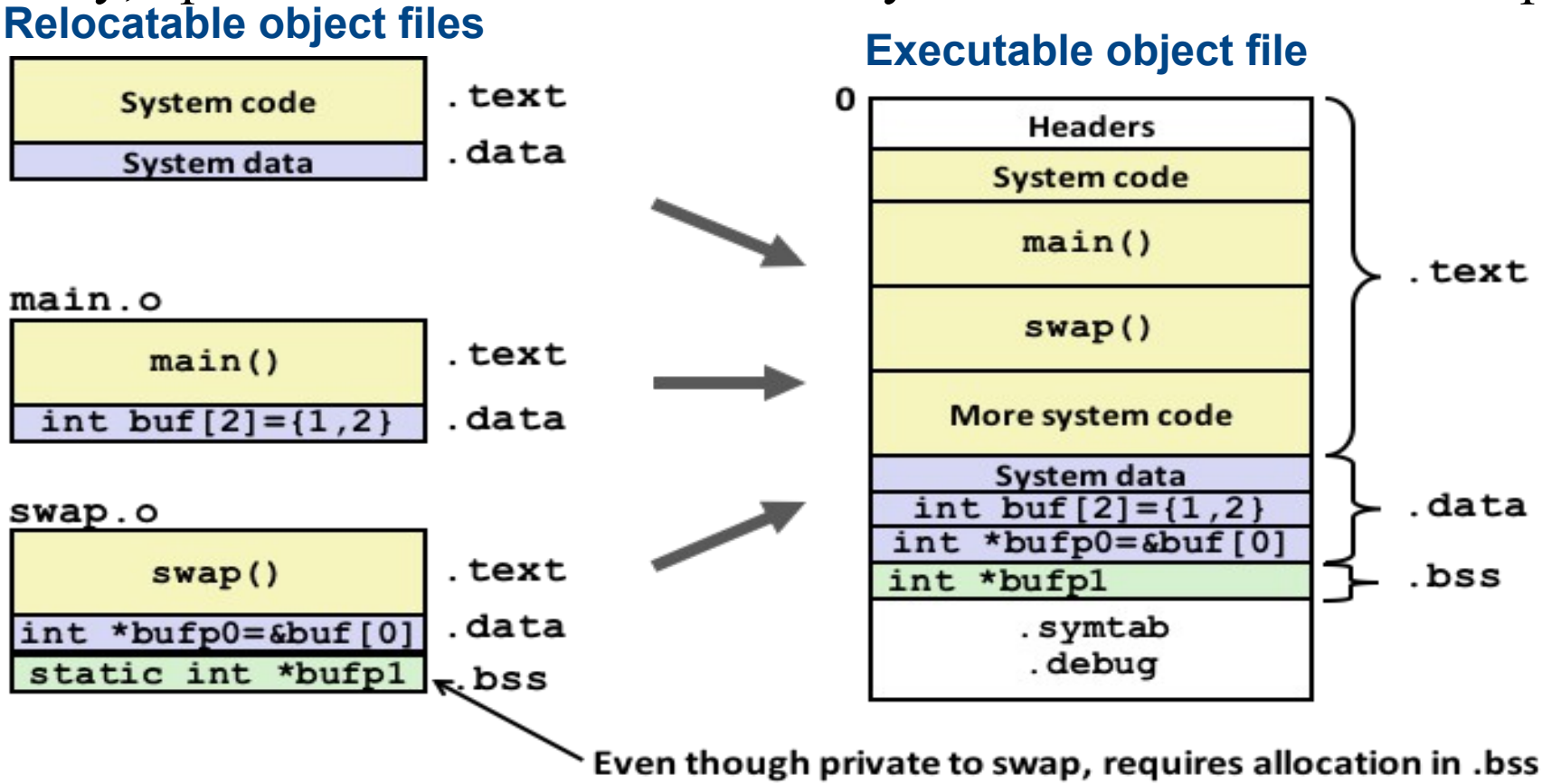
- **Relocation:** Merge code and data sections of multiple object files into the code and data sections of the final executable
- **Symbol Resolution:** Linker associates each symbol reference with exactly one symbol definition

Source code files (main.c, swap.c)
Preprocessing (cpp)
Preprocessed code files (main.i, swap.i)
Compiling (cc)
Assembly code files (main.s, swap.s)
Assembling (as)
Object code files (main.o, swap.o)
gcc main.o swap.o -o myexe
Linking (ld)
Static Library (.a)
Executable file (myexe)
Dynamic Library (.so)
Stored in secondary storage as an executable image
Load Time
Loader
Dynamic Library (.so)
Run Time
Process Address Space in main memory

What Linkers do?

Relocation: For each `.c` file, compilers and assemblers generate code and data sections in each object (`.o`) file that start at address zero

- The linker merges separate code and data sections into single sections
- It then relocates symbols from their relative locations in the .o files to their final absolute memory locations in the executable
- Finally, updates all references to these symbols to reflect their new position



Proof of Concept (Relocation)

`03/linking/link/swap.c,main.c`

Linker Symbols

In the context of linker there are three different kinds of symbols:

1. **Global Symbols:** Symbols that are defined in one module and can be referenced by other modules are called global symbols
2. **External Symbols:** Global symbols that are referenced by a module, but are defined in some other module. Normally declared with `extern` keyword

Non static functions and non static global variables fall in above two categories

3. **Local Symbols:** Symbols that are defined and referenced exclusively by a single module. For example, any global variable or function declared with the `static` keyword is private to that module

Linker Symbols (cont...)

Global

```
int buf[2] = {1, 2};

int main()
{
   swap();
   return 0;
}
```

main.c

External

Global

External

Local

```
extern int buf[];

int *bufp0 = &buf[0];
static int *bufp1;

void swap()
{
   int temp;

   bufp1 = &buf[1];
   temp = *bufp0;
   *bufp0 = *bufp1;
   *bufp1 = temp;
}
```

Global

Linker knows nothing of temp

swap.c

Source : Computer Systems “A Programmer's Perspective”

Proof of Concept (Symbol Resolution)

`03/linking/link/swap.o, main.o`

What Linkers do? (cont...)

Symbol Resolution: Symbol definitions are stored by compiler in symbol table, which is an array of **`structs`**, shown below

Linker associates each **symbol reference** with exactly one **symbol definition**

```
typedef struct{
   int name;     /*string table offset*/
   int value;    /*section offset address of the symbol*/
   int size;     /*object size in bytes*/
   char type:4, /*object, func, section, srcfile*/
        binding:4;/*local or global*/
   char section;/*section header index*/
} Elf_Symbol;
```

What if there are two symbol definitions with the same name?

Linker Symbol Rules

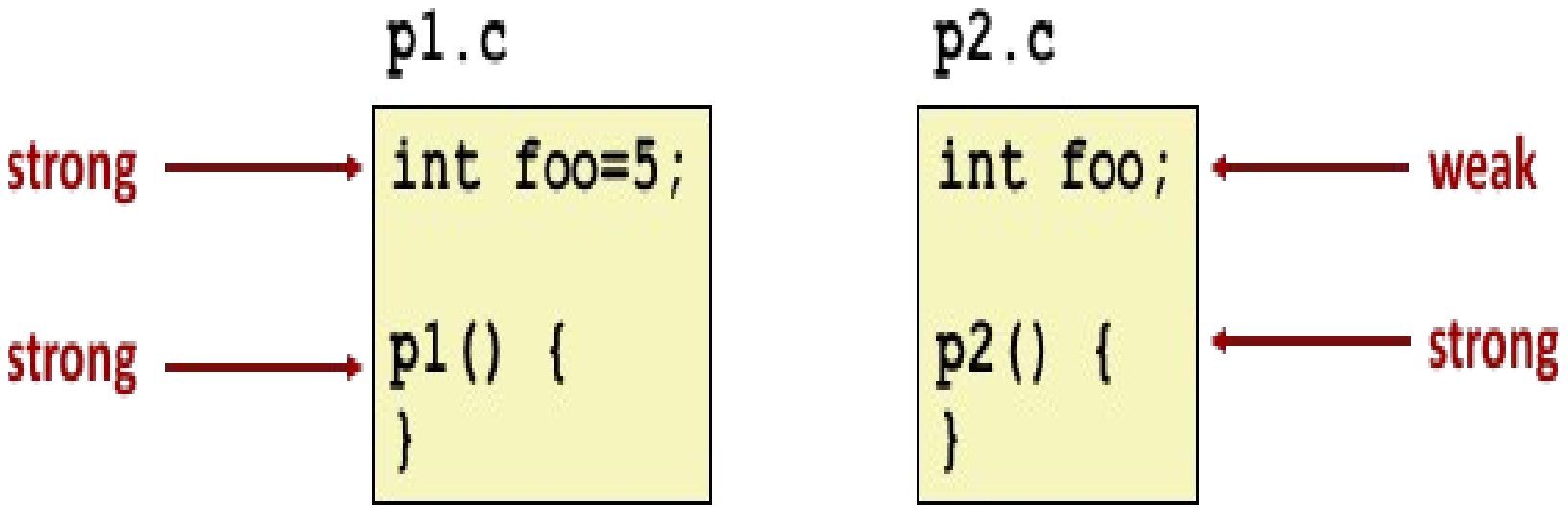
- The linker resolves symbol references by associating each reference with exactly one symbol definition from the symbol tables of its input relocatable object files
- Symbol resolution is straightforward for references to local symbols that are defined and referenced in a single module. However, resolving references to global symbols that are defined in some other module and referenced in some other is trickier
- When the compiler encounters a symbol that is not defined in the current module, it assumes that it is defined in some other module, generates a linker symbol table entry, and leaves it for the linker to handle
- For example, the opposite code file will compile without a hitch, however, the linker terminates when it cannot resolve the reference to function `foo`

```
void foo();
int main(){
    foo();
    return 0
}
```

Linker Symbol Rules (cont...)

The three types of linker symbols (global, external, and local) are either marked as strong or weak.

1. **Strong Symbols:** Function names and initialized globals
2. **Weak Symbols:** Uninitialized globals



Linker Symbol Rules (cont...)

Keeping in mind the concept of strong and weak symbols, UNIX linkers use the following rules for dealing with multiply-defined symbols.

1. **Rule 1:** Multiple strong symbols are not allowed
2. **Rule 2:** Given a strong symbol and multiple weak symbols, choose the strong symbol
3. **Rule 3:** If there are multiple weak symbols, choose an arbitrary one

Proof of Concept
(Handling Conflicts in Symbol Resolution)

`03/linking/symbols/one...five`

How to avoid Symbol Conflicts

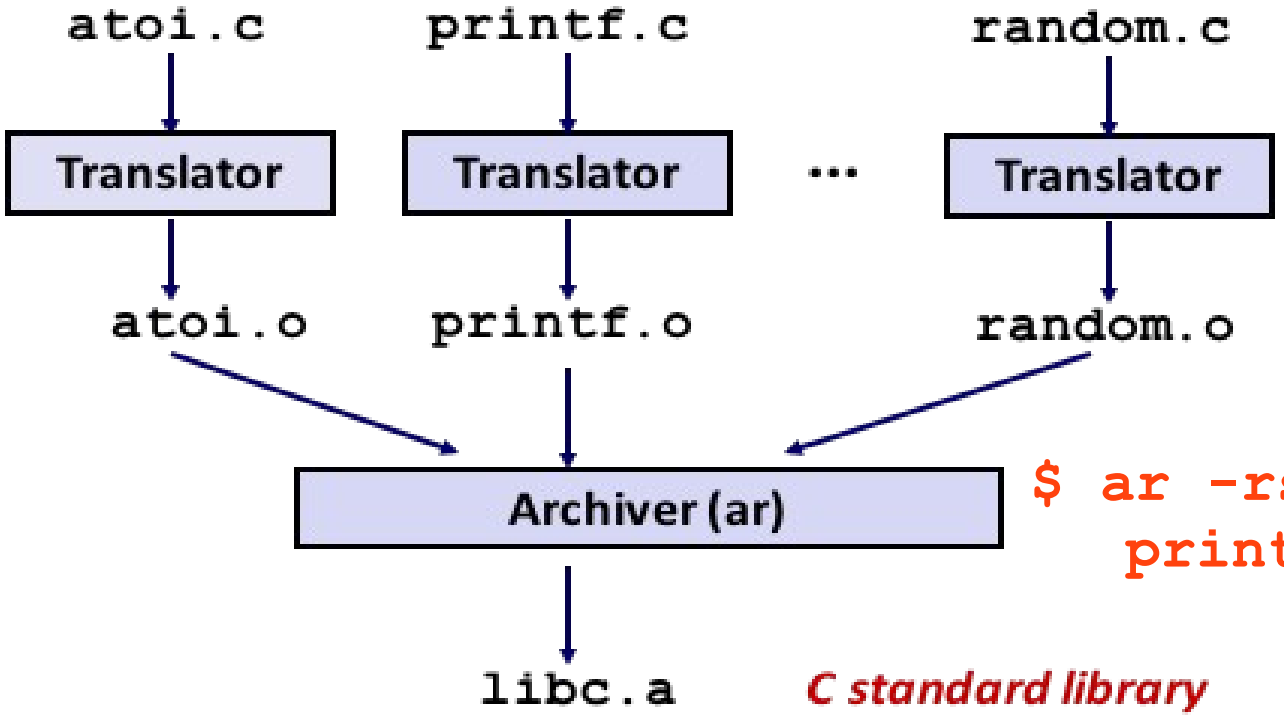
Try to avoid global variables, if you can't

1. Use static keyword with your global variables to make their scope local to that module
2. Always initialize your global variables, thus making them a strong symbol. This way you will get an error while linking, if another strong symbol with the same name exist in another module

Static Libraries
Archives

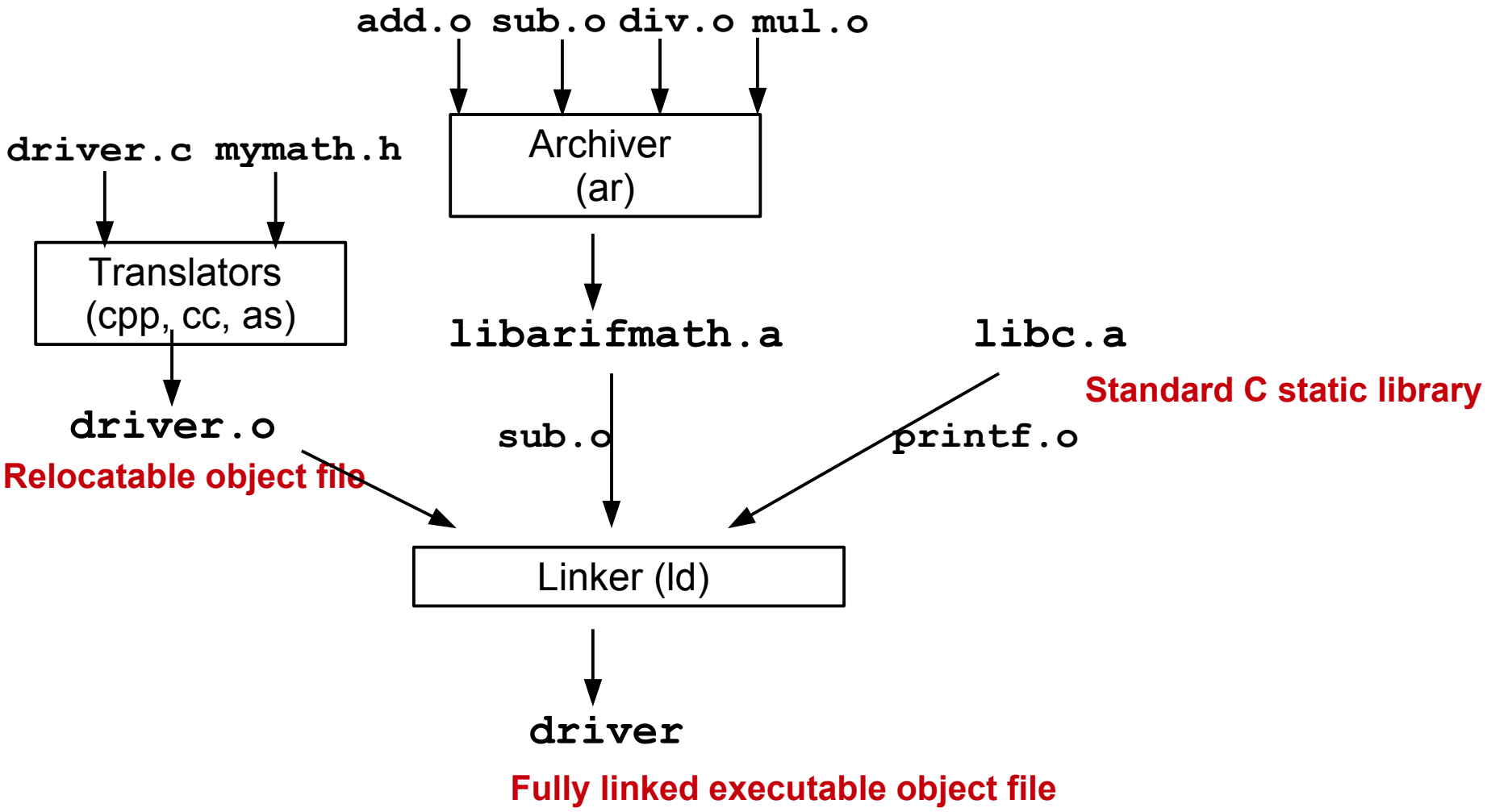
Static Libraries

- Concatenate related relocatable object files into a single file with an index called an archive
- Enhance the linker so that it tries to resolve unresolved external references by looking for the symbols in one or more archives
- If an archive member file resolve reference, link it to the executable
- To make the processes fast **.a** files contains an index for the symbols in all files




```
$ ar -rs libc.a atoi.o  \
   printf.o...random.o
```

Linking with Static Library



The Librarian (ar utility)

ar is a Unix tool also called librarian that allows us to :

1) Create

-r → create a new archive

```
#ar -r libfirst.a file1.o file2.o
```

-q → append an object file to an existing archive

```
#ar -q libfirst.a file3.o
```

-d → delete object modules from an existing archive

```
#ar -d libfirst.a file2.o
```

2)Extract:

-x → extract object modules in your PWD

```
#ar -x /usr/lib/libm.a
```

3)Display:

-t → display table of contents of an archive

```
#ar -t /usr/lib/libm.a
```

Linking with Static Library

Linker's algorithm for resolving external references:

- Scan .o files and .a files in the command line order
- During the scan, keep a list of the current unresolved references
- As each new .o or .a file, obj, is encountered, try to resolve each unresolved reference in the list against the symbols defined in obj
- If any entries in the unresolved list at end of scan, then error

Problem:

- Command line order matters
- Put libraries at the end of the command line

Limitations of Static Libraries

Limitations of static linking:

- The size of executable is large
- Duplication in the executables stored on disk
- Duplication in the executables running in memory. Suppose you are executing ten C-programs all of them using the `scanf` functions. So ten copies of `scanf` functions will be there in memory.
- Minor bug fixes of system libraries require each application to be explicitly relinked

Modern solution: Shared Libraries

- Object files that contain code and data that are loaded and linked into an application dynamically, at either load-time or run-time
- In UNIX world they are called shared objects (.so)
- In MS world they are called dynamic link libraries or dlls

Dynamic Libraries
Shared Objects

Shared libraries

- A shared library is similar to static library because it is also a group of object files however a shared library is different from a static library as the linker and loader both behave differently to a dynamic library.
- A code that can be loaded and executed at any address without being modified by the linker is known as position-independent code. The **`-fPIC`** option to **`gcc`** specifies that the compiler should generate position-independent code. This is necessary for shared libraries, since there is no way of knowing at link time where the shared library code will be located in memory.
- Steps to create a shared library are given below:

Step1: Compile each .c file with -fPIC flag to create object files.

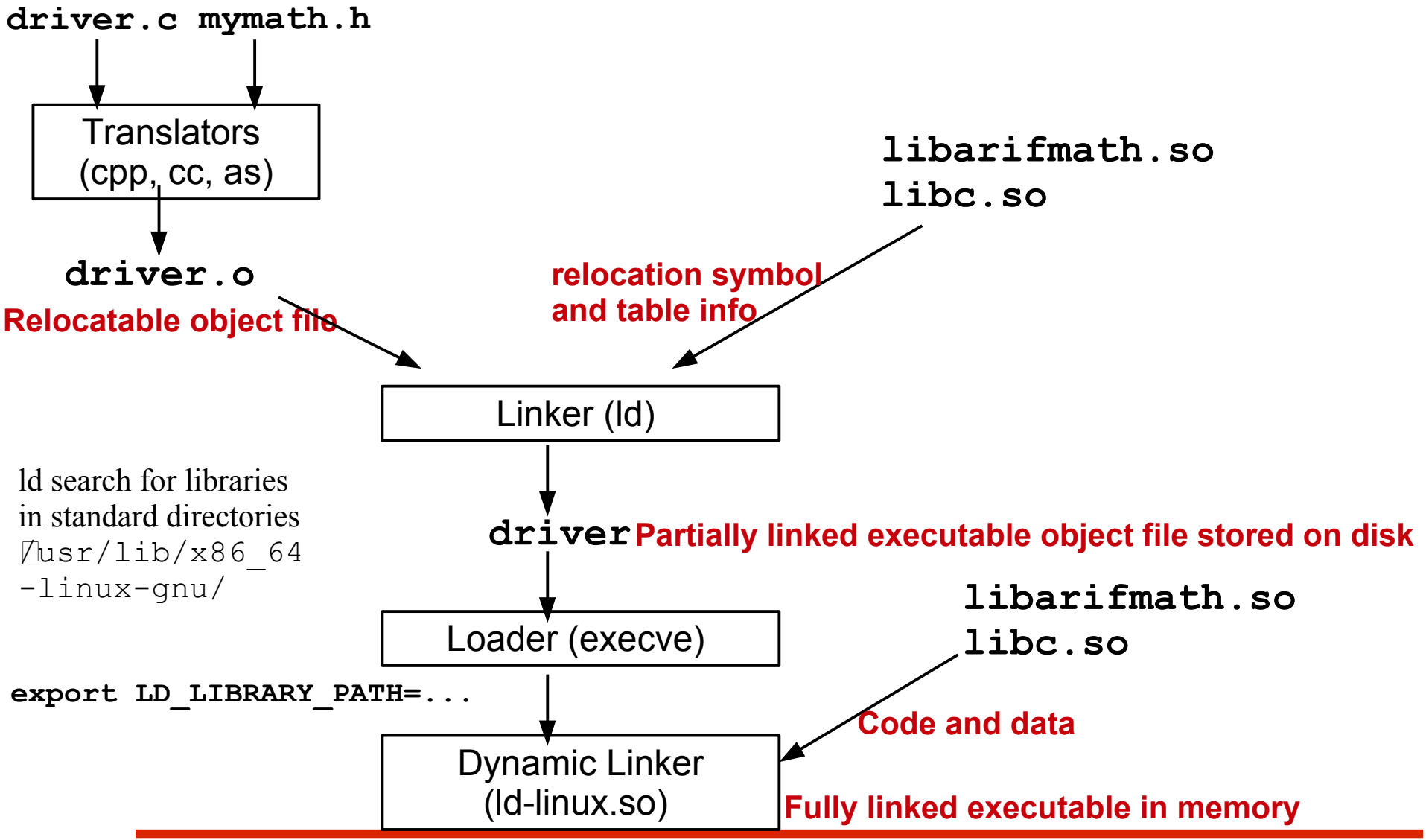
```
$gcc -c -fPIC myadd.c mysub.c mydiv.c mymul.c
```

Step2: Produce a shared object which can then be linked with other objects to form an executable

```
$gcc -shared myadd.o mysub.o mydiv.o mymul.o -o libarifmath.so
```

Dynamic Linking at Load Time

```
$gcc -c -fPIC myadd.c mysub.c mydiv.c mymul.c
$gcc -shared myadd.o mysub.o mydiv.o mymul.o -o libarifmath.so
```



Things To Do

If you have problems visit me in counseling hours. . . .