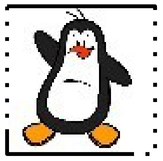


Video Lecture # 03
Linkers (Behind the Curtain)
Creating your own Libraries

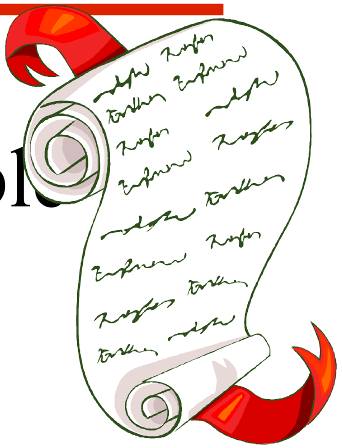
Course: SYSTEM PROGRAMMING

Instructor: Arif Butt

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

Today's Agenda

- Review of Linking process
- Merging Relocatable Object Files into Executabl
- Understanding Linker relocation process
- Understanding Linker symbol resolution
- Creating and using your own Static Libraries
- Creating and using your own Dynamic Libraries



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

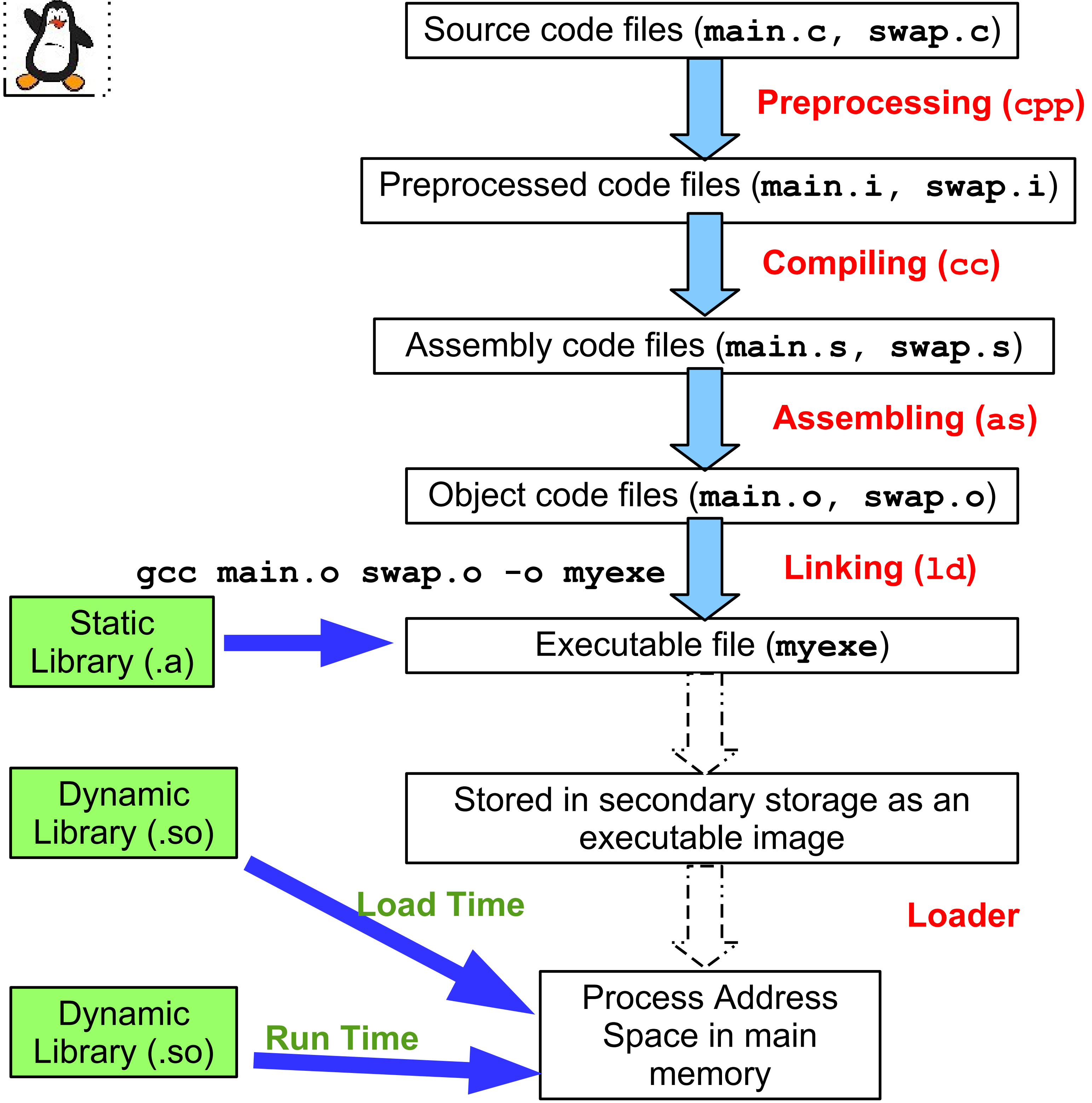
- **Modularity:** Programs can be written as a collection of smaller source files rather than one monolithic mass. We can build libraries of common function like the standard C library `/usr/lib/x86_64-linux-gnu/libc.a`

- **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

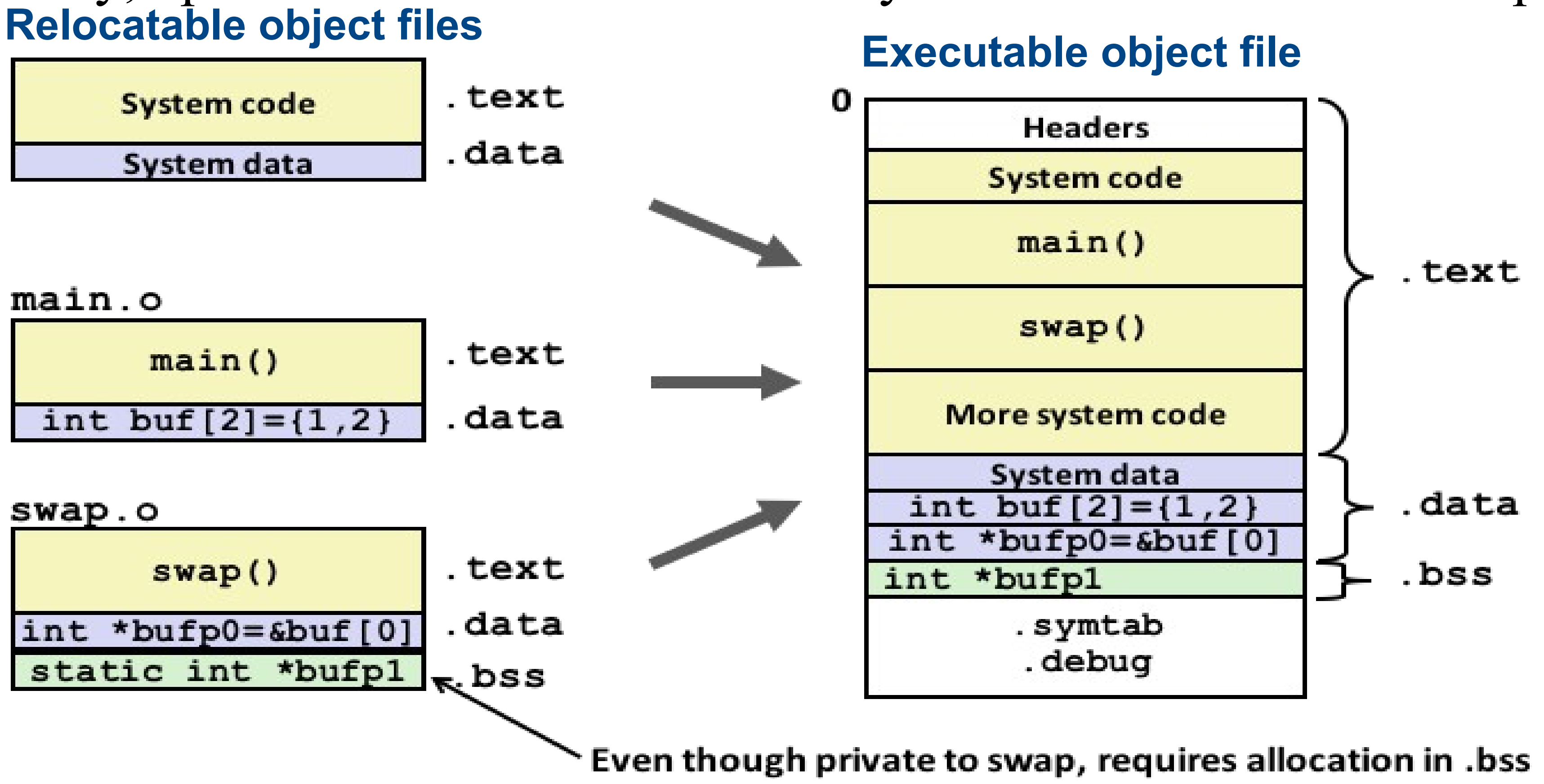
- **Symbol Resolution:** Linker associates each symbol reference with exactly one symbol definition


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)
Stored in secondary storage as an executable image
Dynamic Library (.so)
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



Linker Symbols

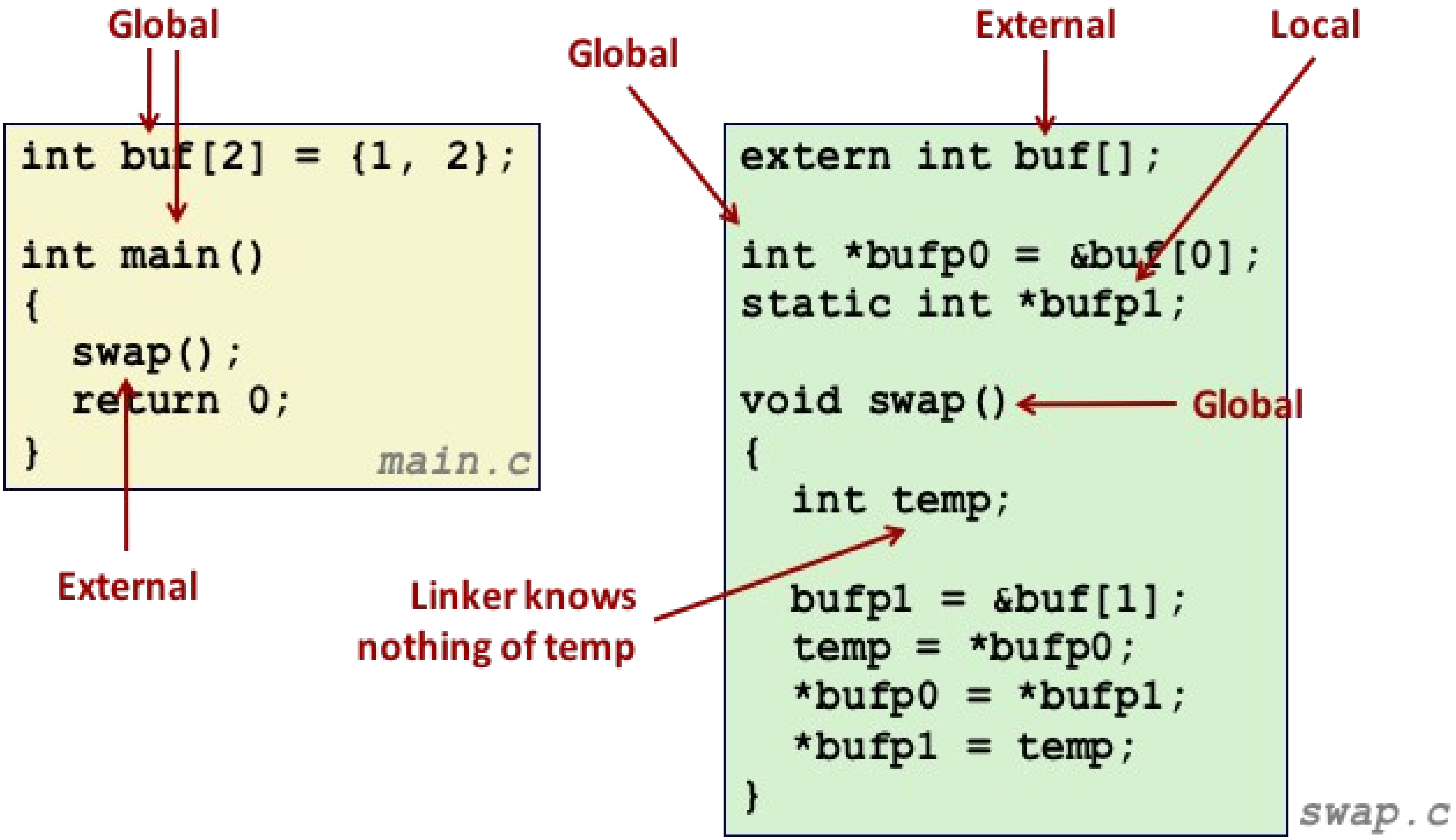
In the context of linker there are three different kinds of symbols:

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...)



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

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

main.c

```
extern int buf[];

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

void swap()
{
  int temp;

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

swap.c

Source : Computer Systems “A Programmer's Perspective”

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

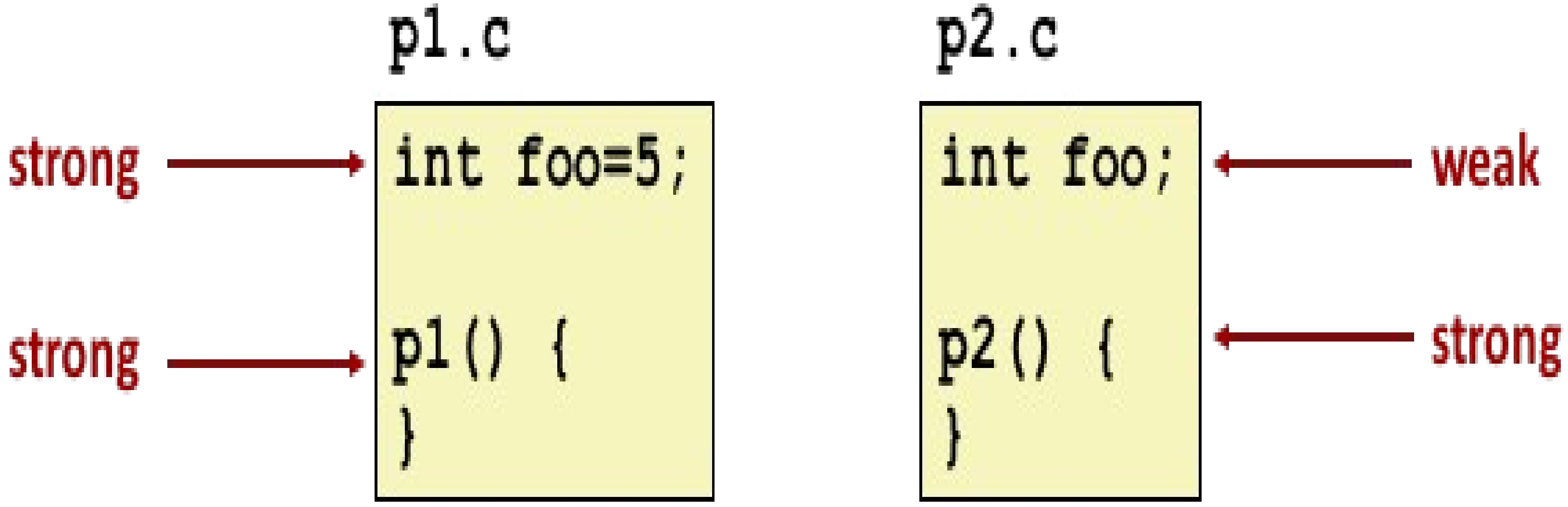
- The linker resolves symbol references by associating each reference with exactly one symbol definition from the symbol tables of its input relocatable object files
- Symbol resolution is straightforward for references to local symbols that are defined and referenced in a single module. However, resolving 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



```
int foo=5;

p1() {
}
```

```
int foo;

p2() {
}
```

Linker Symbol Rules (cont...)

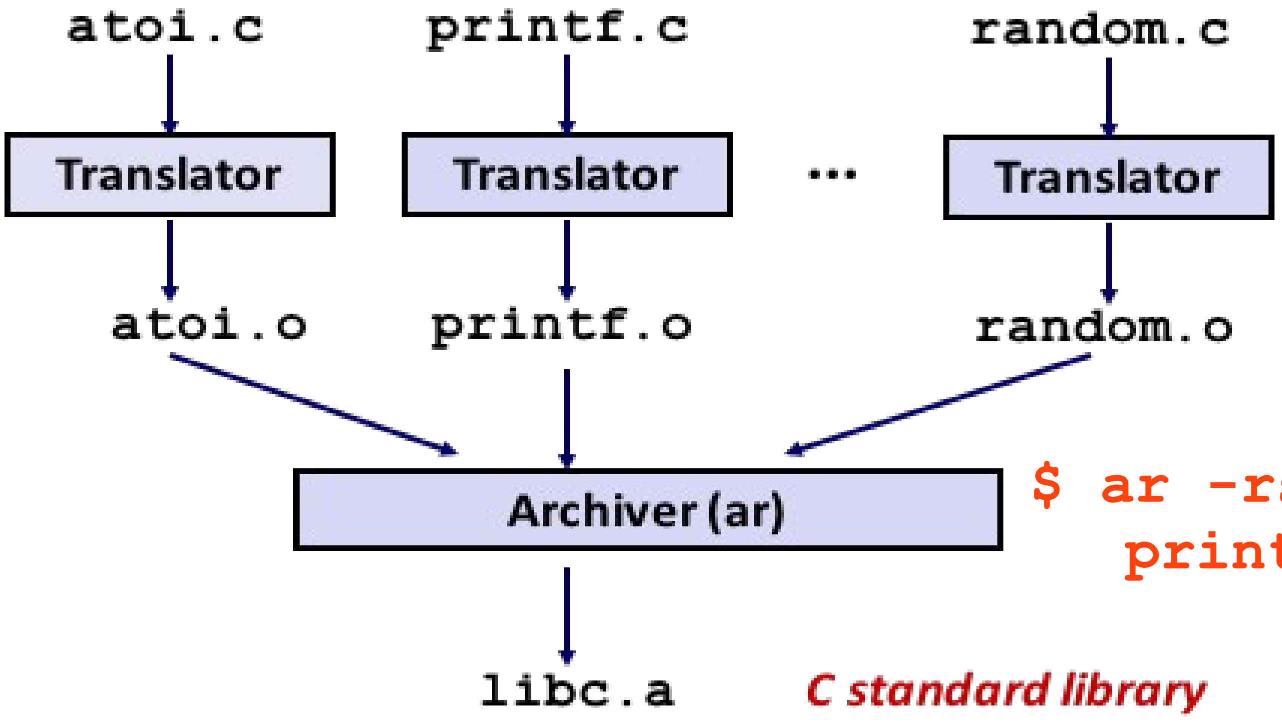
Keeping in mind the concept of strong and weak symbols, UNIX linkers use the following rules for dealing with multiply-defined symbols.

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

Static Libraries Archives

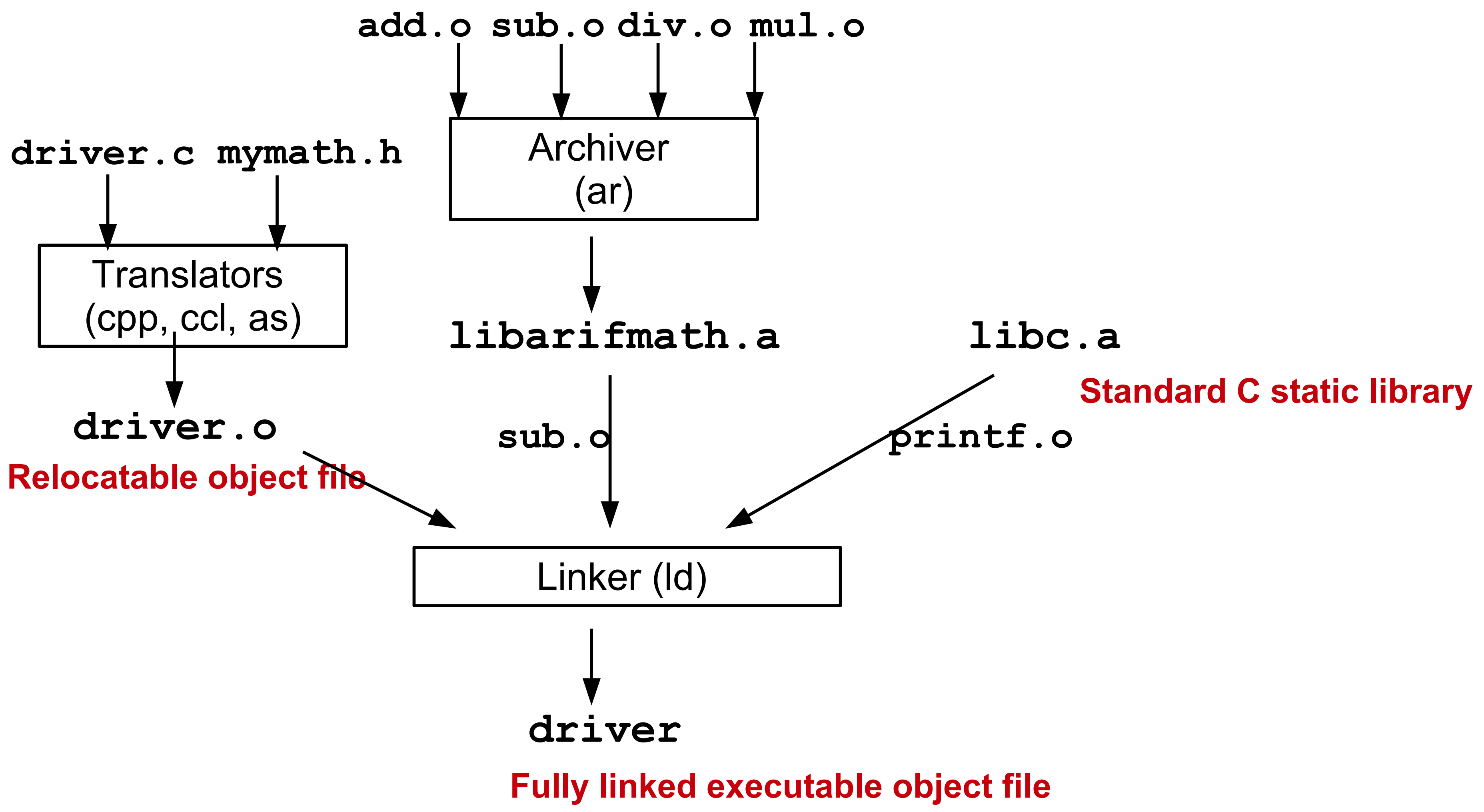
Static Libraries

- 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.

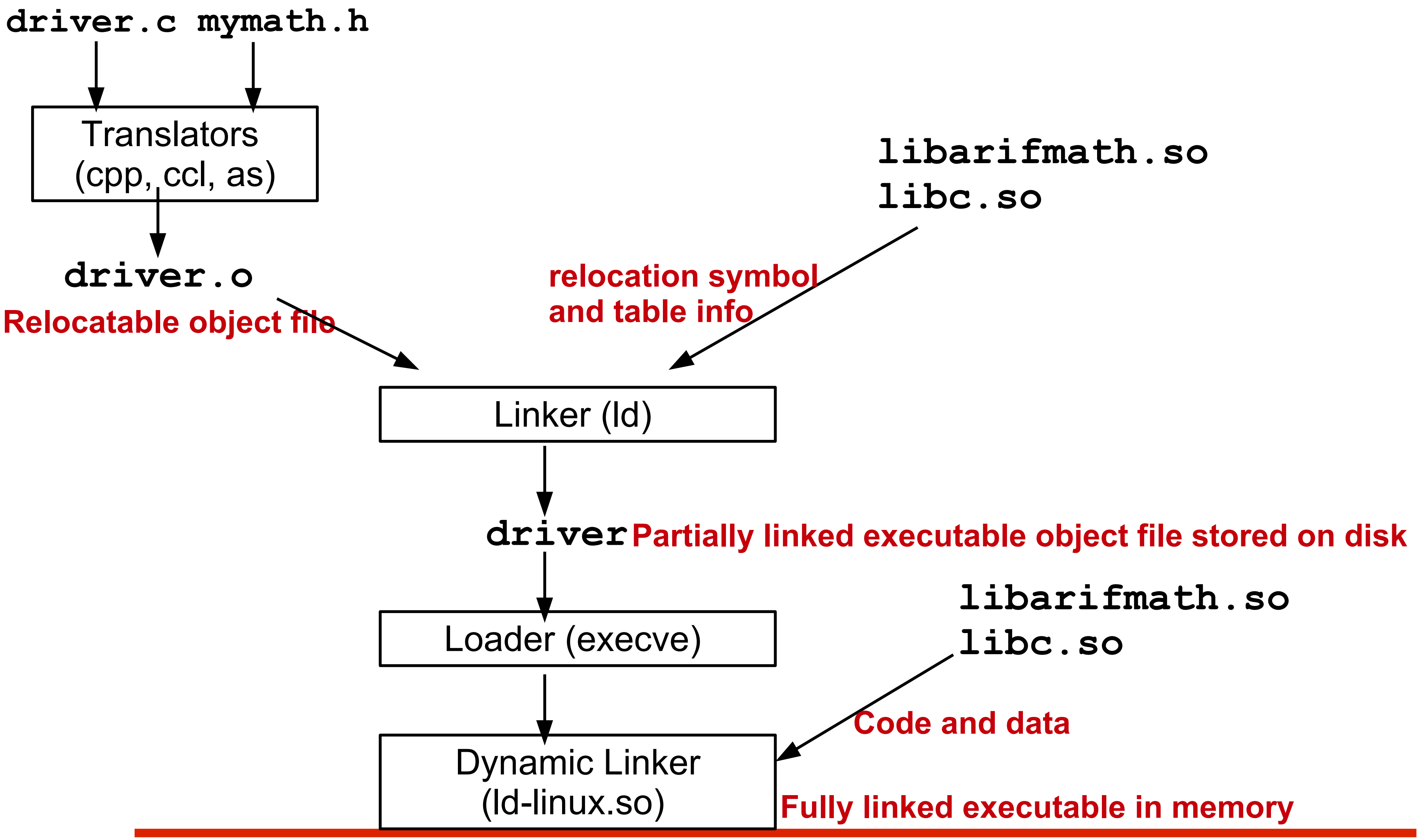
```
$gcc -c -fPIC myadd.c mysub.c mydiv.c mymul.c
```

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
```


driver.c
mymath.h
Translators (cpp, ccl, as)
driver.o
Relocatable object file
libarifmath.so
libc.so
relocation symbol and table info
Linker (ld)
driver
Partially linked executable object file stored on disk
Loader (execve)
libarifmath.so
libc.so
Code and data
Dynamic Linker (ld-linux.so)
Fully linked executable in memory

Things To Do

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