Linking

(#) Linking is the process of collecting and combining various pieces of code and data into a single file that can be loaded into memory and executed.
Linking can be performed at:
1. Compile time
2. Loading time
3. Runtime.

* Running Example

(a) main.c — code/link/main.c

```
1  int sum(int *a, int n);
2
3  int array[2] = {1, 2};
4
5  int main()
6  {
7      int val = sum(array, 2);
8      return val;
9  }
```

code/link/main.c

(b) sum.c — code/link/sum.c

```
1  int sum(int *a, int n)
2  {
3      int i, s = 0;
4
5      for (i = 0; i < n; i++) {
6          s += a[i];
7      }
8      return s;
9  }
```

code/link/sum.c

Figure 7.1 Example program 1. The example program consists of two source files, `main.c` and `sum.c`. The `main` function initializes an array of `int`s, and then calls the `sum` function to sum the array elements.

* Static Linking

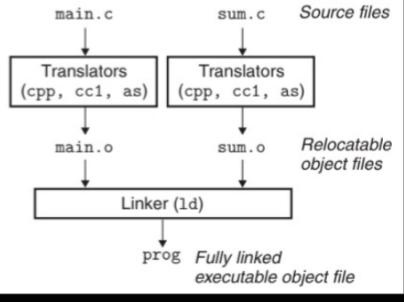


Figure 7.2 Static linking. The linker combines relocatable object files to form an executable object file prog.

Shell Commands.

```
linux> gcc -Og -o prog main.c sum.c
```

`cpp` [*other arguments*] `main.c /tmp/main.i`

Next, the driver runs the C compiler (`cc1`), which translates `main.i` into an ASCII assembly-language file `main.s`:

`cc1 /tmp/main.i -Og` [*other arguments*] `-o /tmp/main.s`

Then, the driver runs the assembler (`as`), which translates `main.s` into a binary *relocatable object file* `main.o`:

`as` [*other arguments*] `-o /tmp/main.o /tmp/main.s`

The driver goes through the same process to generate `sum.o`. Finally, it runs the linker program `ld`, which combines `main.o` and `sum.o`, along with the necessary system object files, to create the binary *executable object file* `prog`:

`ld -o prog` [*system object files and args*] `/tmp/main.o /tmp/sum.o`

To run the executable `prog`, we type its name on the Linux shell's command line:

```
linux> ./prog
```

The shell invokes a function in the operating system called the *loader*, which copies the code and data in the executable file `prog` into memory, and then transfers control to the beginning of the program.

Q. What exactly are the following things?

① GCC (GNU Compilation Collection)
↳ It's a driver program, doesn't do compilation
↳ It calls the
1. cpp
2. cc1
3. as
4. ld

② Cpp (C Pre Processor)
↳ preprocessor for C language
↳ Input .c file
↳ Output .i file

③ cc1
↳ actual compiler
↳ Input .i file
↳ Output .as file.

④ as/gas (Gnu assembler)
↳ assembles for C
↳ Input .as file
↳ Output .o file.

⑤ Ld (
↳ Linker
↳ Input .o
↳ Output .exe.

(#) Some other compiler
FqS1 → fortan
JC1 → Java
CC1plus → C++.

↳ Static linker takes collection of relocatable object files and command line arguments and generate as a fully linked executable.
↳ To build an executable the linker must perform 2 tasks.
① Symbol resolution : associate one symbol reference with exactly one symbol definition.

> *Step 1. Symbol resolution.* Object files define and reference *symbols*, where each symbol corresponds to a function, a global variable, or a *static variable* (i.e., any C variable declared with the `static` attribute). The purpose of symbol resolution is to associate each symbol *reference* with exactly one symbol *definition*.

② Relocation :
↳ compilers and assemblers generates codes and data section that start at 0 address.
↳ linker relocates these sections by associating a memory location to each symbol definition and modifying all the references to those symbols.

(#) Object files are merely collections of blocks of bytes
↳ Some blocks contains program code
↳ Some contains program data
↳ Some contains data structure that guide the linker and loader.

Linker concats all of these blocks together decides runtime location for these concated blocks and modifies their references.

↳

* Object files.

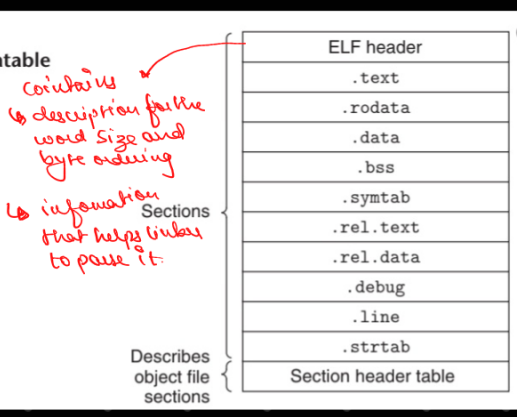
① Relocatable object file: (ROF) → generated by compilers and assemblers
↳ Contains binary code and data in a form that can be combined with other relocatable object files at compile time to create an Executable object file.

② Executable object file (EOF) → linkers generate this.
↳ Contains binary code and data in the form that can be directly copied into memory and executed

③ Shared object file (SOF)
↳ A special type of relocatable object file that can be loaded into memory and linked dynamically at either load or run time.

* Relocatable Object file

Figure 7.3
Typical ELF relocatable object file.



`.text`	The machine code of the compiled program.
`.rodata`	Read-only data such as the format strings in `printf` statements, and jump tables for switch statements.
`.data`	*Initialized* global and static C variables. Local C variables are maintained at run time on the stack and do *not* appear in either the `.data` or `.bss` sections.
`.bss`	*Uninitialized* global and static C variables, along with any global or static variables that are initialized to zero. This section occupies no actual space in the object file; it is merely a placeholder. Object file formats distinguish between initialized and uninitialized variables for space efficiency: uninitialized variables do not have to occupy any actual disk space in the object file. At run time, these variables are allocated in memory with an initial value of zero.
`.symtab`	A *symbol table* with information about functions and global variables that are defined and referenced in the program. Some programmers mistakenly believe that a program must be compiled with the `-g` option to get symbol table information. In fact, every relocatable object file has a symbol table in `.symtab` (unless the programmer has specifically removed it with the `STRIP` command). However, unlike the symbol table inside a compiler, the `.symtab` symbol table does not contain entries for local variables.
`.rel.text`	A list of locations in the `.text` section that will need to be modified when the linker combines this object file with others. In general, any instruction that calls an external function or references a global variable will need to be modified. On the other hand, instructions that call local functions do not need to be modified. Note that relocation information is not needed in executable object files, and is usually omitted unless the user explicitly instructs the linker to include it.
`.rel.data`	Relocation information for any global variables that are referenced or defined by the module. In general, any initialized global variable whose initial value is the address of a global variable or externally defined function will need to be modified.
`.debug`	A debugging symbol table with entries for local variables and typedefs defined in the program, global variables defined and referenced in the program, and the original C source file. It is only present if the compiler driver is invoked with the `-g` option.
`.line`	A mapping between line numbers in the original C source program and machine code instructions in the `.text` section. It is only present if the compiler driver is invoked with the `-g` option.
`.strtab`	A string table for the symbol tables in the `.symtab` and `.debug` sections and for the section names in the section headers. A string table is a sequence of null-terminated character strings.

↳ each Relocatable object module m, has a symbol table that contains info about the symbols that are defined and referenced by m.
↳ Global symbols defined by module m and that can be used by other modules
↳ global symbols that are referenced by m but defined by some other module.
↳ local symbols that are defined and referenced exclusively by module m.

previously we defined that local variables are stored in stack but static local value is assigned space in .data or .bss.

COMMON	Uninitialized global variables
.bss	Uninitialized static variables, and global or static variables that are initialized to zero

* Symbol Resolution

① Gather: collect all symbols (defined + Undefined) from objects files and libraries
② Match: Pair undefined references with definitions
③ check: each symbol has only one definition.

Resolving references to global references is tricky as the definition may be in other object file.
→ may be there are multiple definitions
etc.

↳ How Linkers resolve Replicate Symbol names.
↳ At Compile time, the compiler exports each global symbol to the assembler as strong or weak
① Strong: functions and initialized global variable
② weak: Uninitialized global variables.

Rules.
① Multiple strong symbol with same names ✗
② if weak and strong have same name we choose strong
③ given multiple weak with same, choose any weak.

```c
/* foo1.c */
int main()
{
    return 0;
}
```

```c
/* bar1.c */
int main()
{
    return 0;
}
```

→ multiple strong not allowed.

```c
/* foo2.c */
int x = 15213;

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

```c
/* bar2.c */
int x = 15213;

void f()
{
}
```

```c
/* foo3.c */
#include <stdio.h>
void f(void);

int x = 15213;

int main()
{
    f();
    printf("x = %d\n", x);
    return 0;
}
```

```c
/* bar3.c */
int x;

void f()
{
    x = 15212;
}
```

→ strong

→ local.

no problem with error, but the local variable will overwrite the global variable.

#Static library : a file format that is designed to be used during static linking.

Static linker performs linking using static library.

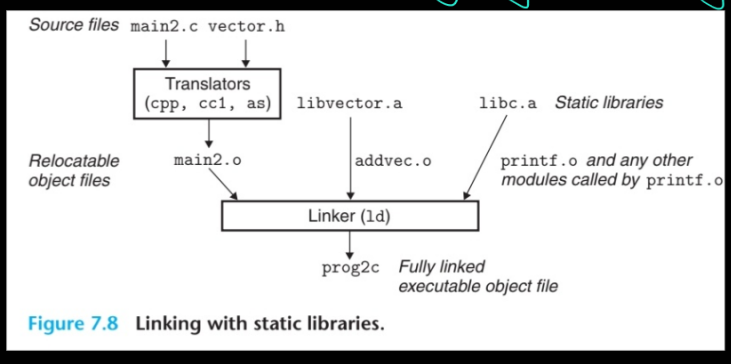


Figure 7.8 Linking with static libraries.

* Relocation

Consists of 2 steps.

① Relocating Sections and Symbol definitions.

↳ Linker merges all sections of the same type into a new aggregate section of new type.

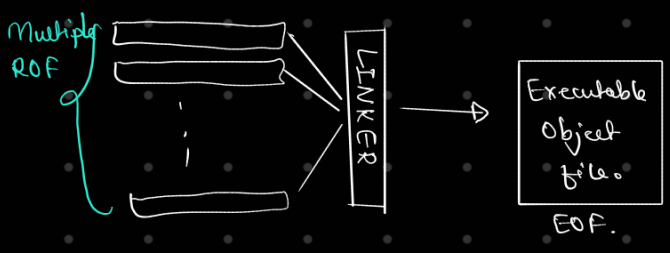
↳ The linker then assigns run time memory addresses to these new sections.

② Relocating Symbol references within sections

↳ Linker modifies every symbol reference in the bodies of code and data so they match with correct run time addresses.

When assembler was generating code and it came up with a reference it does not know what to do at that point so, it puts that reference in a Relocation entry

↳ Relocation step uses this relocation entry to calculate and update references.



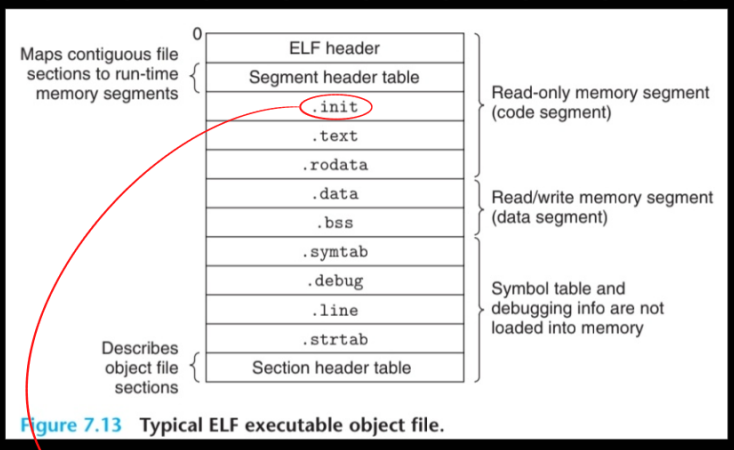


Figure 7.13 Typical ELF executable object file.

↳ defines a small _init func" which will be called when program starts.

* Loading Executable Object file

↳ whenever we run command

linux > ./prog

↳ it assumes prog is an exe as there is no command prog in Linux.

↳ it runs some memory-resident OS code known as loader

↳ loader copies the code and data in EOF from disk into memory and runs the program by jumping to its first instn. or entry point

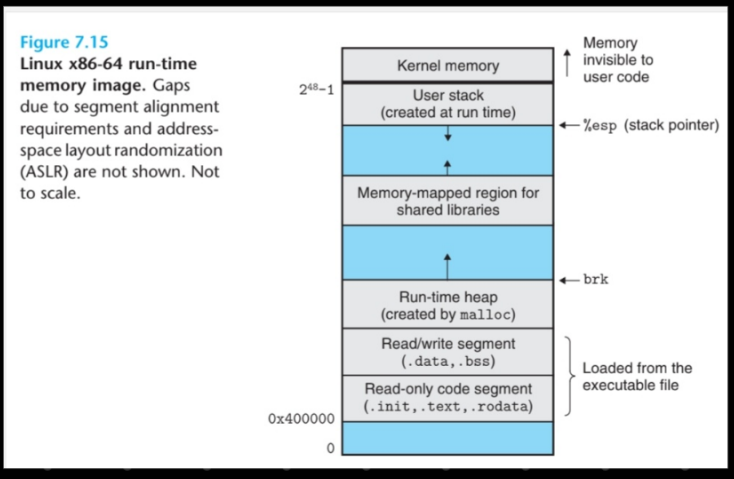
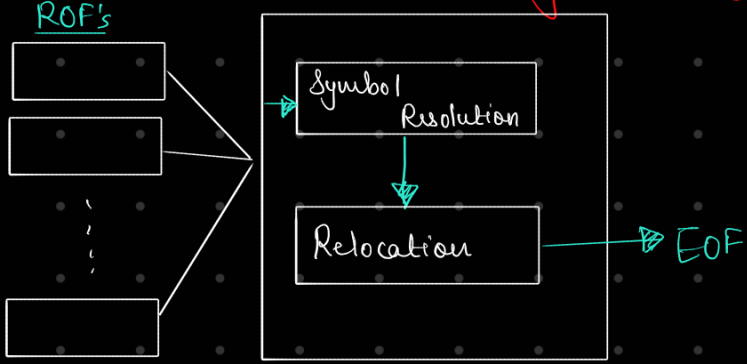


Figure 7.15 **Linux x86-64 run-time memory image.** Gaps due to segment alignment requirements and address-space layout randomization (ASLR) are not shown. Not to scale.

↳ Loader copies the chunks of EOF into code and data segments.

* Actual flow of Static linker (as per my understanding)



Aside How do loaders really work?

Our description of loading is conceptually correct but intentionally not entirely accurate. To understand how loading really works, you must understand the concepts of *processes*, *virtual memory*, and *memory mapping*, which we haven't discussed yet. As we encounter these concepts later in Chapters 8 and 9, we will revisit loading and gradually reveal the mystery to you.

For the impatient reader, here is a preview of how loading really works: Each program in a Linux system runs in the context of a process with its own virtual address space. When the shell runs a program, the parent shell process forks a child process that is a duplicate of the parent. The child process invokes the loader via the `execve` system call. The loader deletes the child's existing virtual memory segments and creates a new set of code, data, heap, and stack segments. The new stack and heap segments are initialized to zero. The new code and data segments are initialized to the contents of the executable file by mapping pages in the virtual address space to page-size chunks of the executable file. Finally, the loader jumps to the `_start` address, which eventually calls the application's `main` routine. Aside from some header information, there is no copying of data from disk to memory during loading. The copying is deferred until the CPU references a mapped virtual page, at which point the operating system automatically transfers the page from disk to memory using its paging mechanism.

* Issues with Static Libraries

↳ needs to be maintained and updated periodically.
↳ we copy general codes like printf, scanf etc into our code which is a waste of space resources.

* Dynamic Linker

↳ Shared library is an object module that at either run time or load time, can be loaded at an arbitrary memory address and linked with the program in memory
↳ This process is known as dynamic linking

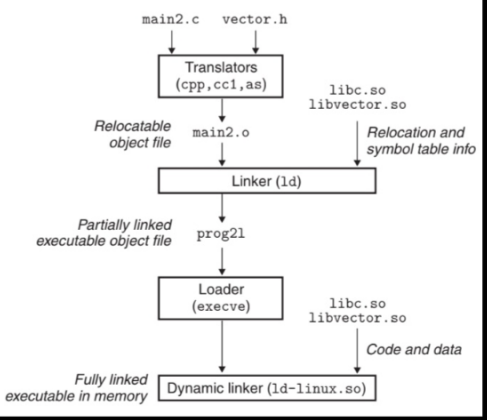


Figure 7.16 Dynamic linking with shared libraries.

Shared libraries are shared in 2 different ways

① In every system, there is exactly one .so file for a particular library → math library.
↳ libc.so.6, libm.so.6
↳ System call wrappers
↳ <math.h> only contains the definition for func". actual code is here.

② a single copy of .text section of a shared lib in memory can be shared by multiple processes.

* Position Independent Code (PIC)

↳ how do we allocate memory space to .so file

① We can allocate a dedicated chunk of memory for .so files only.
↳ this is a wastage of resource if our program is not using .so files.

② Compile the shared files such that the code can be loaded without needing any relocation. this code is known as PIC
↳ for this approach we exploit the fundamentals of 'offset'. we dont need to assign every reference a new memory, we can assign them based on the offset of the starting point.
↳ this is done with the help of global offset Table (GOT)
↳ The GOT contains 8 byte entry for each global data that is referenced by object modules.

* PIC function calls

There are again 2 approaches for knowing the run time address of function calls from shared object module
① we generate relocation table for every function at the time of linking
↳ Inefficient, what if only one func" is to be called
② lazy binding : assign referring functions address only when it is first called.