Linking & Loading

(Slides include materials from *Operating System Concepts*, 7th ed., by Silbershatz, Galvin, & Gagne and from *Modern Operating Systems*, 2nd ed., by Tanenbaum)

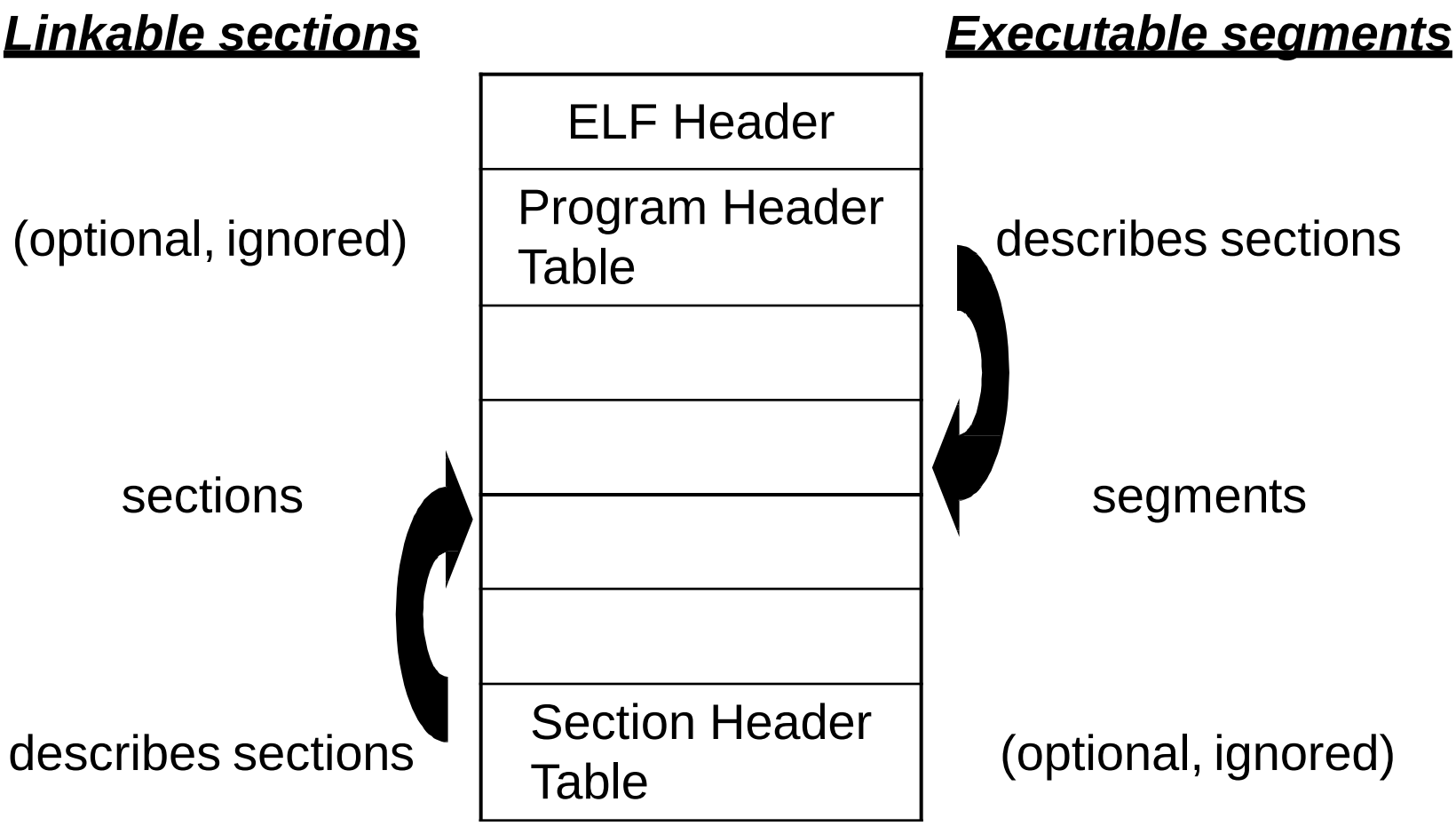
What happens to your program …

…after it is compiled, but before it can be run?

Executable files

- Every OS expects executable files to have a specific format
  - *Header info*
    - Code locations
    - Data locations
  - Code & data
  - *Symbol Table*
    - List of *names* of things defined in your program and where they are located within your program.
    - List of *names* of things defined elsewhere that are used by your program, and where they are used.

Example: ELF Files (x86/Linux)



Example

```
#include <stdio.h>


int main () {

 printf ("hello,
 world\n")


}
```

- Symbol defined in your program and used elsewhere
  - `main`
- Symbol defined elsewhere and used by your program
  - `printf`

Example

```
#include <stdio.h>
extern int errno;

int main () {

 printf ("hello, world\n")

 <check errno for errors>
}
```

- Symbol defined in your program and used elsewhere
  - `main`
- Symbol defined elsewhere and used by your program
  - `printf`
  - `errno`

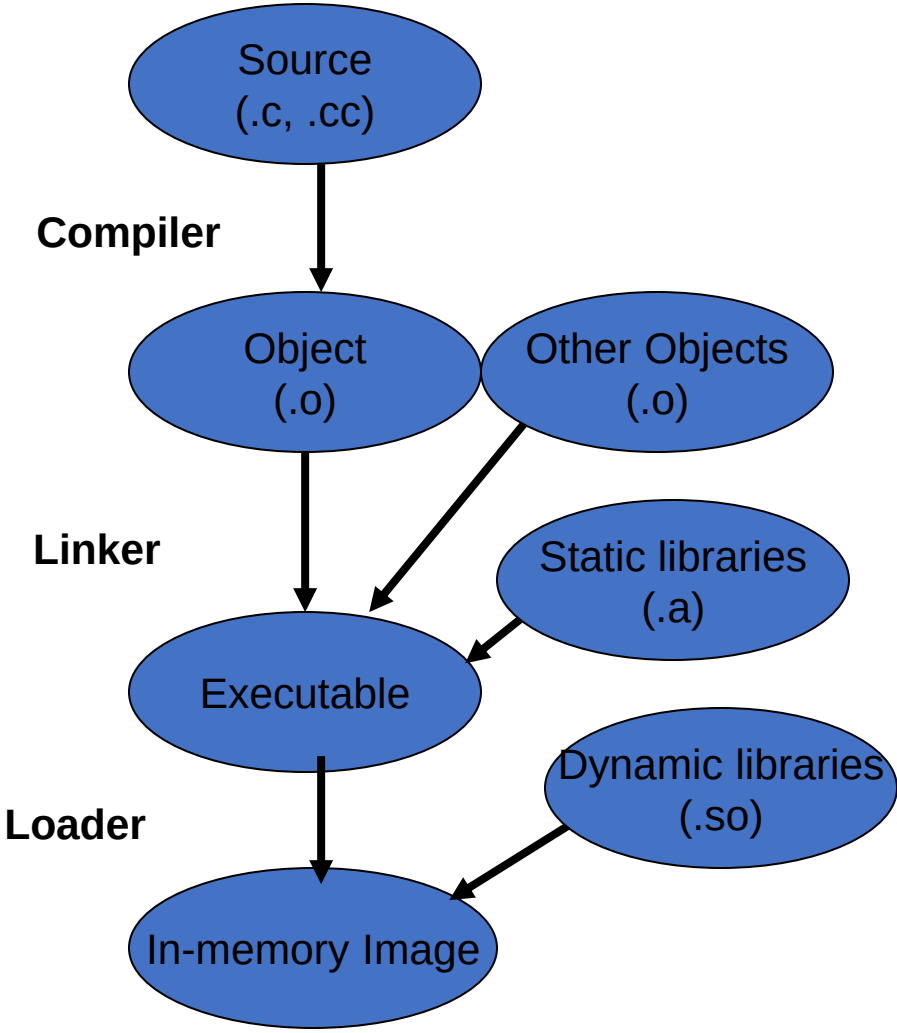
Two-step operation
(in most systems)

- *Linking*: Combining a set of programs, including library routines, to create a *loadable image*
  a) Resolving symbols defined within the set
  b) Listing symbols needing to be resolved by loader

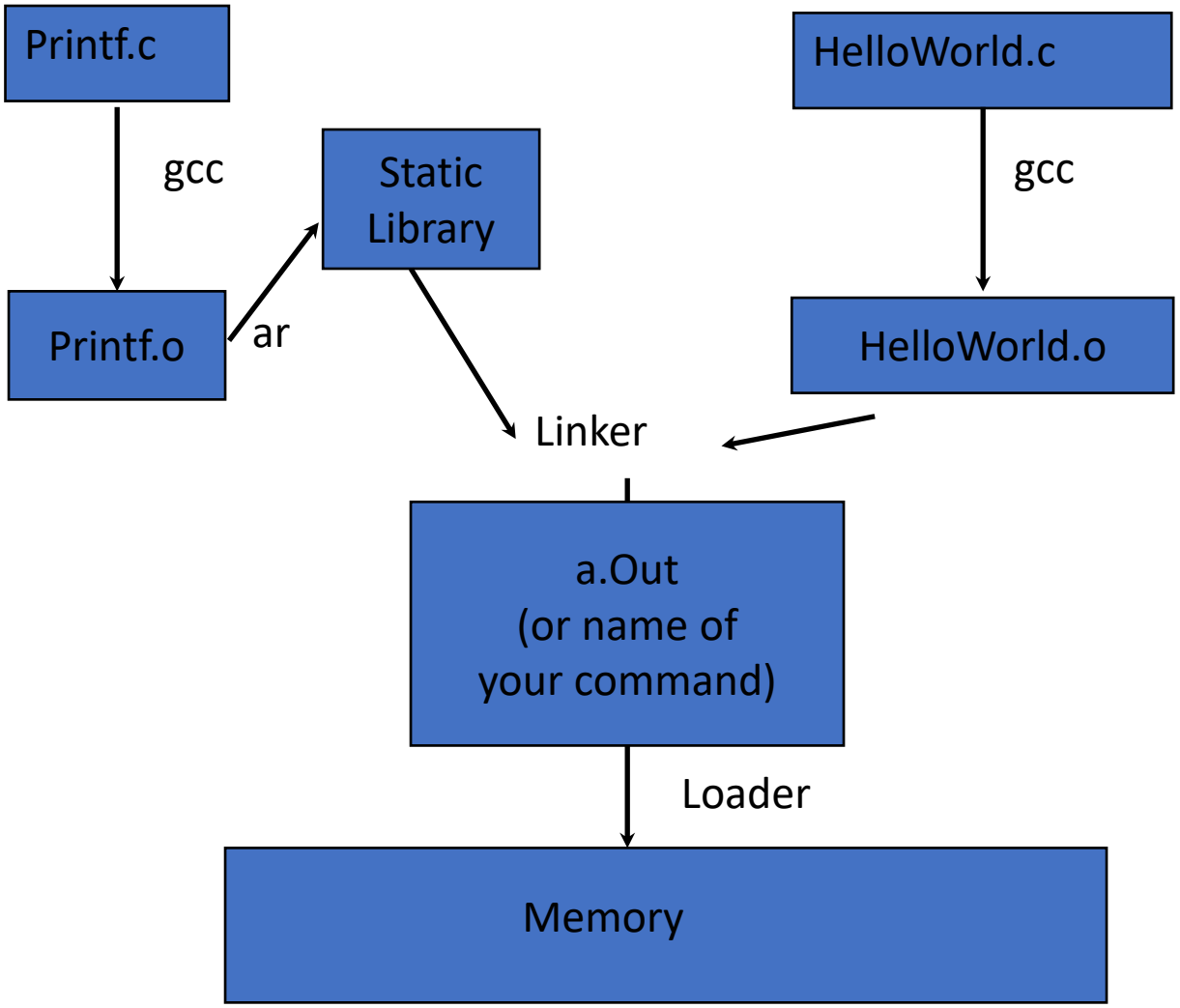
- *Loading:* Copying the loadable image into memory, connecting it with any other programs already loaded, and updating addresses as needed
  - (In Unix) interpreting file to initialize the process address space
  - (in all systems) kernel image is special (own format)

From source code to a process

- *Binding* is the act of connecting *names* to *addresses*
- Most compilers produce *relocatable object code*
  - Addresses relative to *zero*
- The linker combines multiple object files and library modules into a single executable file
  - Addresses also relative to *zero*
- The Loader reads the executable file
  - Allocates memory
  - Maps addresses within file to memory addresses
  - Resolves names of dynamic library items



Static Linking and Loading



Classic Unix

- Linker lives inside of *cc* or *gcc* command
- Loader is part of *exec* system call
- Executable image contains *all* object and library modules needed by program
- Entire image is loaded at once

- Every image contains its own copy of common library routines
- Every loaded program contain duplicate copy of library routines

Loading

- It loads a program file for execution
- Two approaches
  - Static loading
  - Dynamic loading
- Advantages of dynamic loading
  - Better memory-space utilization; unused routine is never loaded.
  - Useful when large amounts of code are needed to handle infrequently occurring cases

Static Library and Loading

- Static libraries are .a files. All the code relating to the library is in this file, and it is directly linked into the program at compile time. A program using a static library takes copies of the code that it uses from the static library and makes it part of the program.

Dynamic Loading

- Routines in dynamic libraries (.so files) are not loaded until it is called
- Better memory-space utilization; unused routines are never loaded.
- Useful when large amounts of code are needed to handle infrequently occurring cases.
- *In computer programming, routine and subroutine are general and nearly synonymous terms for any sequence of code that is intended to be called and used repeatedly during the executable of a program*

Program-controlled Dynamic Loading

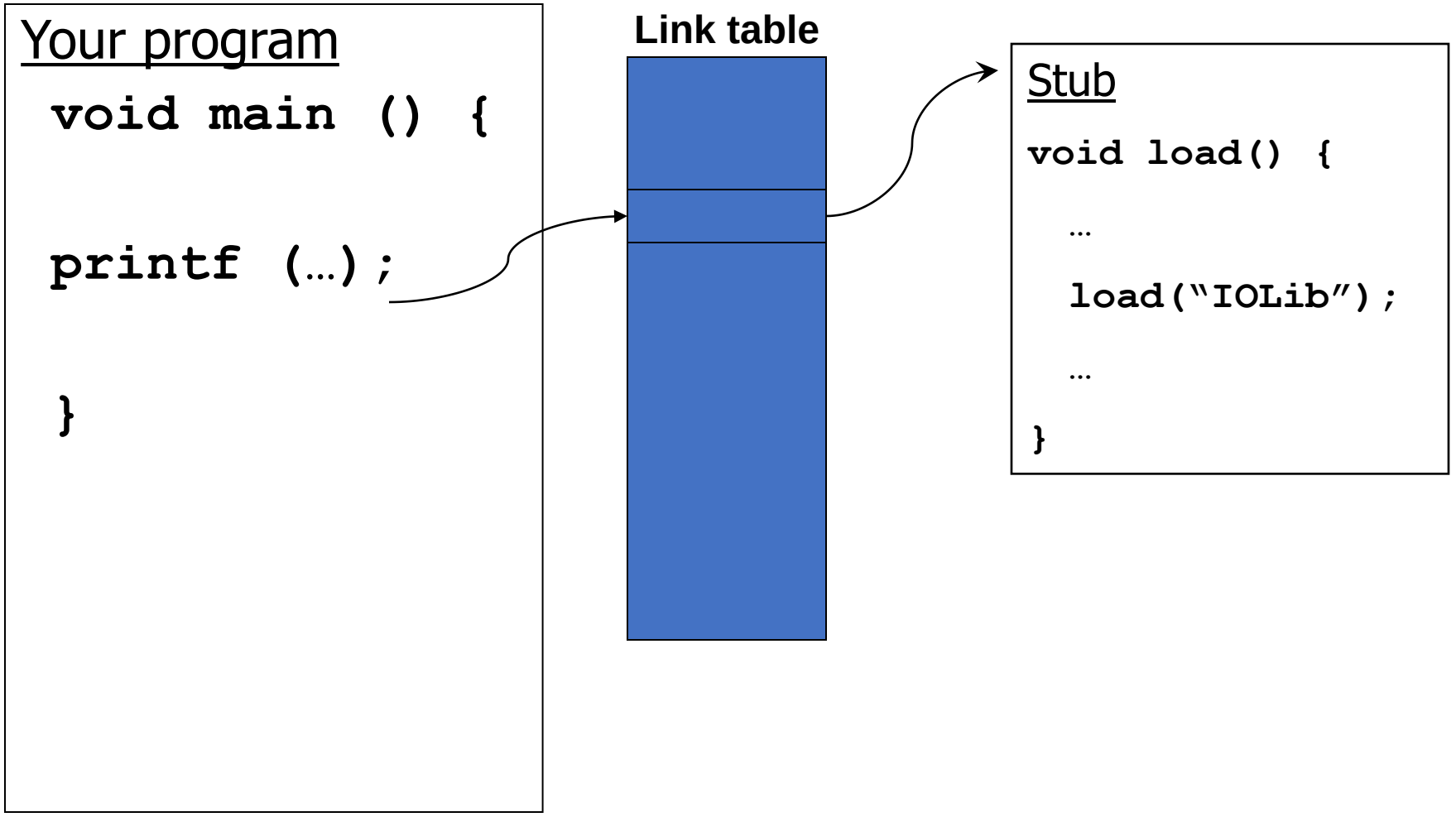
- Requires:
  - A *load* system call to invoke loader (not in classical Unix)
  - ability to leave symbols unresolved and resolve at run time (not in classical Unix)
- E.g.,

```
void myPrintf (**arg) {
 static int loaded = 0;
 if (!loaded ) {
  load ("printf");
  loaded = 1;
 printf(arg);
 }
}
```

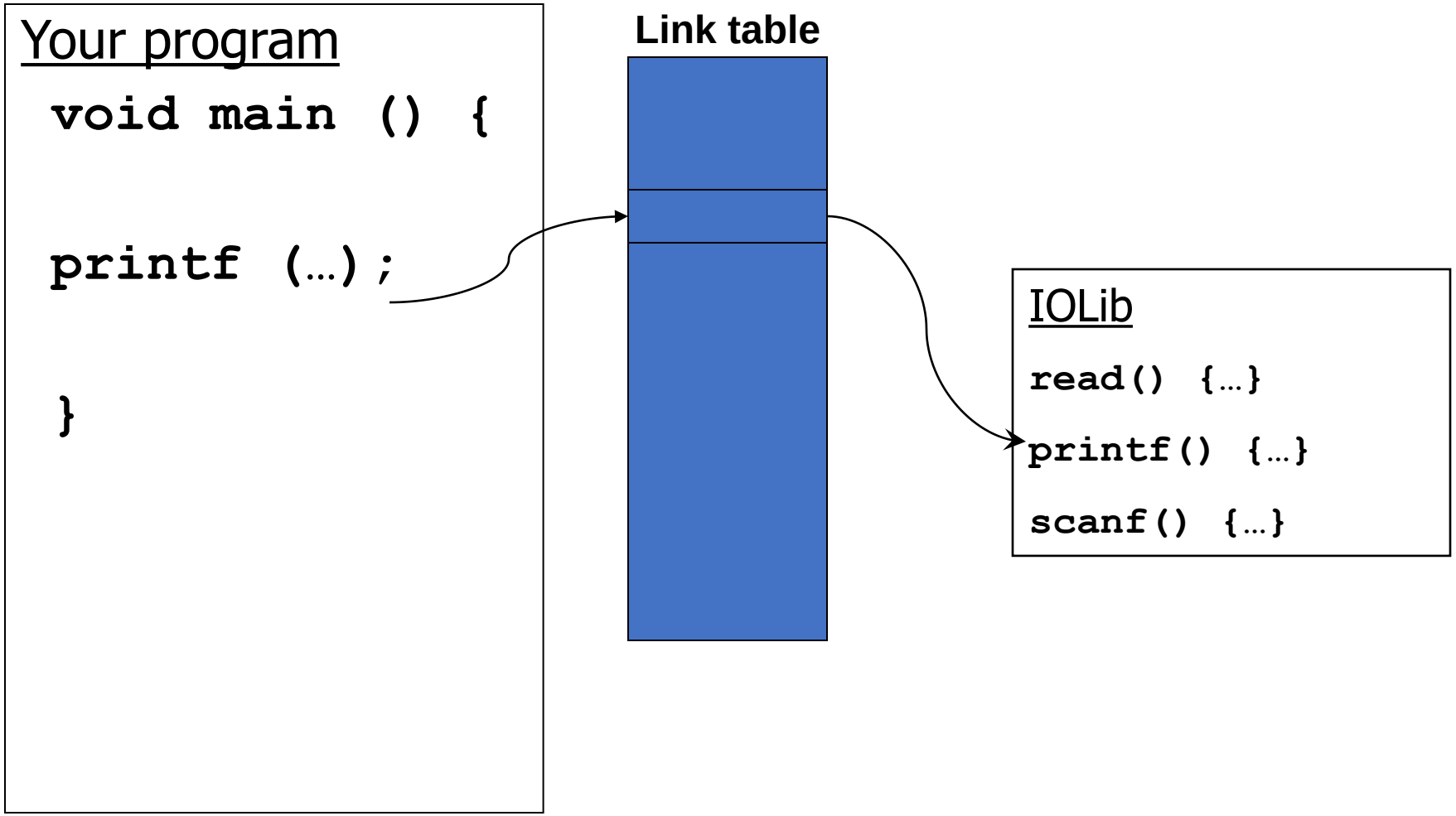
Linker-assisted Dynamic Loading

- Programmer marks modules as “dynamic” to linker
- For function call to a dynamic function
  - Call is indirect through a *link table*
  - Each link table entry is initialized with address of small *stub* of code to locate and load module.
  - When loaded, loader replaces link table entry with address of loaded function
  - When unloaded, loader restores table entry with stub address
  - Works only for *function calls*, not *static data*

Example – Linker-assisted loading
(before)



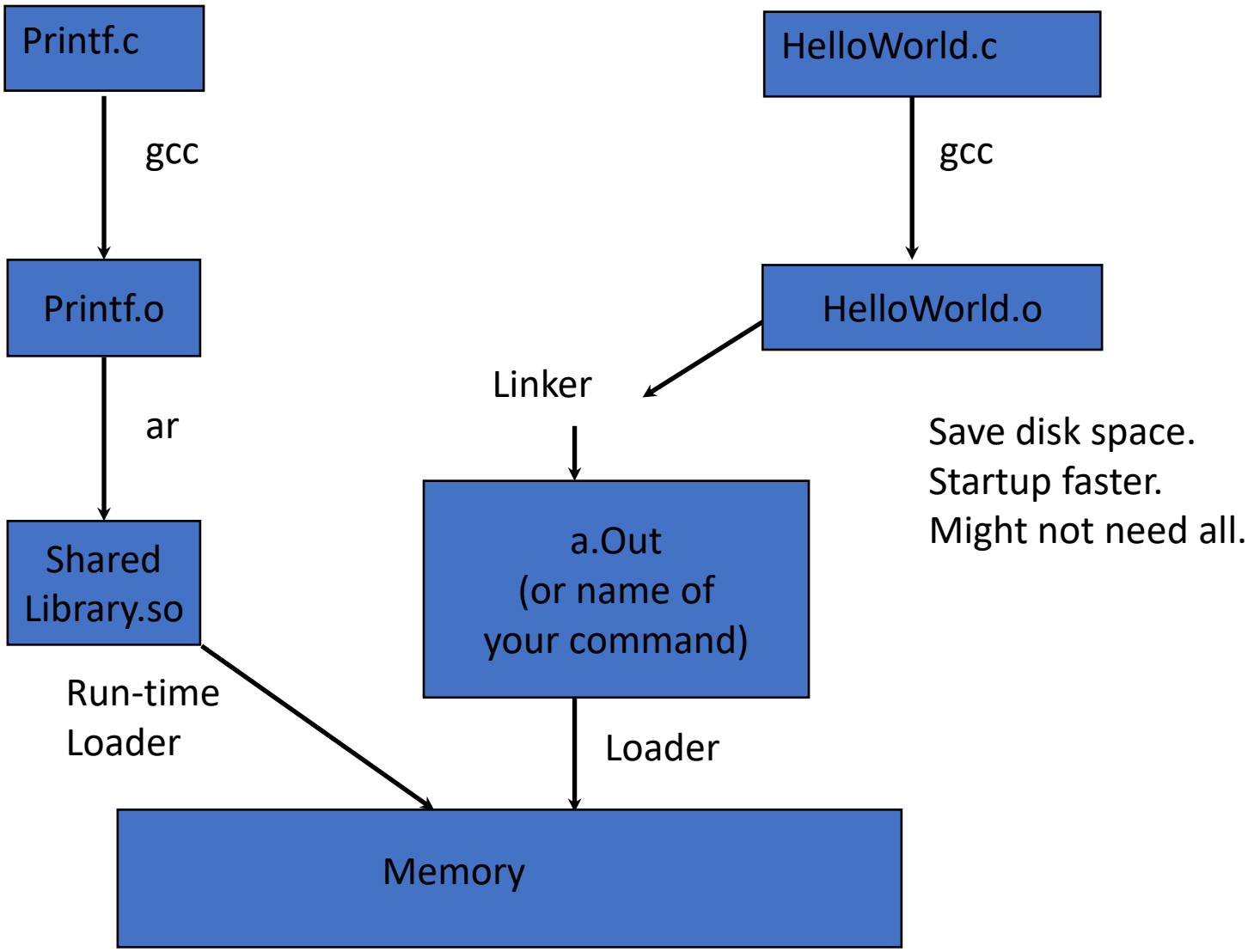
Example – Linker-assisted loading
(after)



Shared libraries

- Libraries designated as “shared”
  - .so, .dll, etc.
- *Linker* sets up symbols to be resolved at runtime
- *Loader:* Is library already in memory?
  - If yes, *map* into new process space
    - “map,” an operation to be defined later in course
  - If not, load and then *map*

Run-time Linking/Loading



Dynamic Linking

- Complete linking postponed until execution time.
- *Stub* used to locate the appropriate memory-resident library routine.
- Stub replaces itself with the address of the routine, and executes the routine.
- Operating system needs to check if routine is in address space of process
- Dynamic linking is particularly useful for libraries.

Dynamic Linking

- Dynamic vs. static linking
  - $ 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 (asm code)
    - 1688756 hello-static
- If you are the sys admin, which do you prefer?

Advantages of Dynamic Linking

- The executable is smaller (it not include the library information explicitly),
- 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 codes can access the same . so at the same time (saves memory)

Disadvantages of Dynamic Linking

- Performance hit ~10%
- Need to load shared objects (once)
- Need to resolve addresses (once or every time)
- What if the necessary dynamic library is missing?
- Could have the library, but wrong version

Unix Dynamic Objects (.so)

- Compiler Options (cont)
  - -static link only to static (.a=archive) libraries
  - -shared if possible, prefer shared libraries over static
  - -nostartfiles skip linking of standard start files, like /usr/lib/crt[0,1].o, /usr/lib/crti.o, etc
- Linker Options (gcc gives unknown options to linker)
  - -l lib (default naming convention liblib.a)
  - -L lib path (in addition to default /usr/lib and /lib)
  - -s strip final executable code of symbol and relocation tables

Loader

- An integral part of the OS
- Resolves addresses and symbols that could not be resolved at link-time
- May be small or large
  - Small: Classic Unix
  - Large: Linux, Windows XP, etc.
- May be invoke explicitly or implicitly
  - Explicitly by stub or by program itself
  - Implicitly as part of *exec*
- *Loader searching path*
  - *Defined by environment parameter*
    - *LD_LIBRARY_PATH*
  - *Or some predefined searching path*
    - *Current path, /usr/lib, /usr/lib64, etc.*