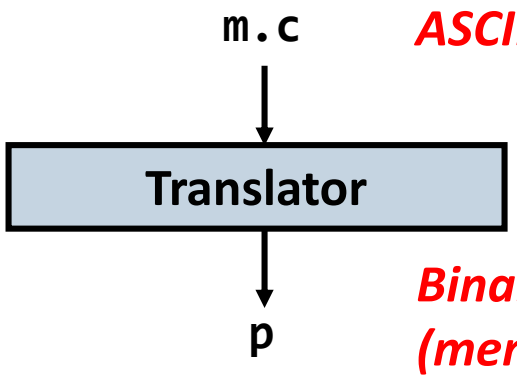
Linking

Jin-Soo Kim (*jinsookim@skku.edu*)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu

Basic Program Translation

- A simplistic program translation scheme



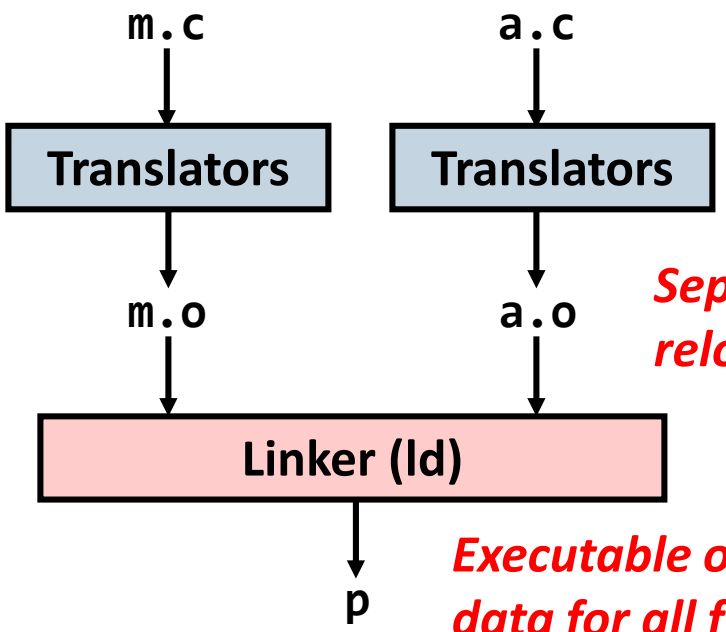
ASCII source file

Binary executable object file (memory image on disk)

  - Problems
    - Efficiency: small change requires complete recompilation
    - Modularity: hard to share common functions (e.g. `printf`)
  - Solution
    - Static linker

Program Translation with Linker

- A better scheme using a static linker



Separately compiled relocatable object files

Executable object file (contains code and data for all functions defined in `m.c` and `a.c`)

General Program Translation

- Compiler driver coordinates all steps in the translation and linking process
  - Typically included with each compilation system (gcc)
  - Invokes preprocessor (cpp), compiler (cc1), assembler (as), and linker (ld)
  - Passes command line arguments to appropriate phases

```
$ gcc -O2 -v -o p main.c swap.c
  cpp [args] main.c /tmp/main.i
  cc1 /tmp/main.i -O2 [args] -o /tmp/main.s
  as [args] -o /tmp/main.o /tmp/main.s
  <similar process for swap.c>
  ld -o p [system obj files] /tmp/main.o /tmp/swap.o
$
```

Why Linkers?

- Modularity
  - Program can be written as a collection of smaller source files, rather than one monolithic mass
  - Can build libraries of common functions (e.g. **`libc`**, **`libm`**)
- Efficiency in time
  - Change one source file, compile, and then relink
  - No need to recompile other source files
- Efficiency in space
  - Common libraries can be aggregated into a single file
  - Yet executable files and running memory image contain only code for the functions they actually use

What Do Linkers Do?

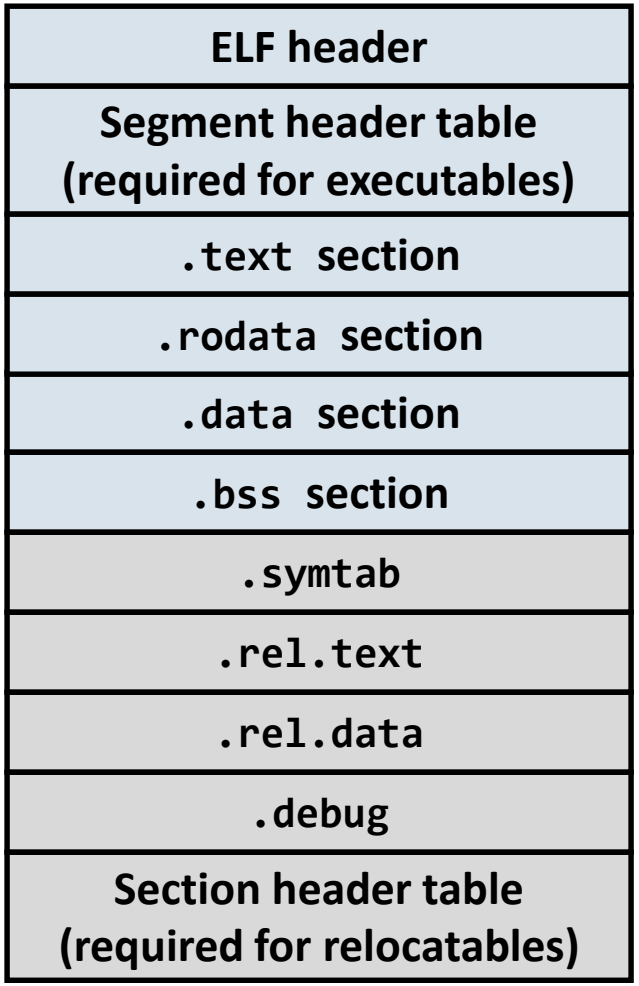
- Merge object files
  - Multiple relocatable (`.o`) object files → a single executable object file that can be loaded and executed by the loader
- Resolve external references
  - External reference: reference to a symbol defined in another object file
- Relocate symbols
  - Relocate symbols from their relative locations in the `.o` files to new absolute positions in the executable
  - Update all references to these symbol to reflect their new positions (either code or data)

ELF

- Executable and Linkable Format (ELF)
  - Standard library format for object files
  - Derived from AT&T System V Unix
    - Later adopted by BSD Unix variants and Linux
  - One unified format for
    - Relocatable object files (`.o`)
    - Executable object files
    - Shared object files (`.so`)
  - Generic name: ELF binaries
  - Better support for shared libraries than old `a.out` formats.

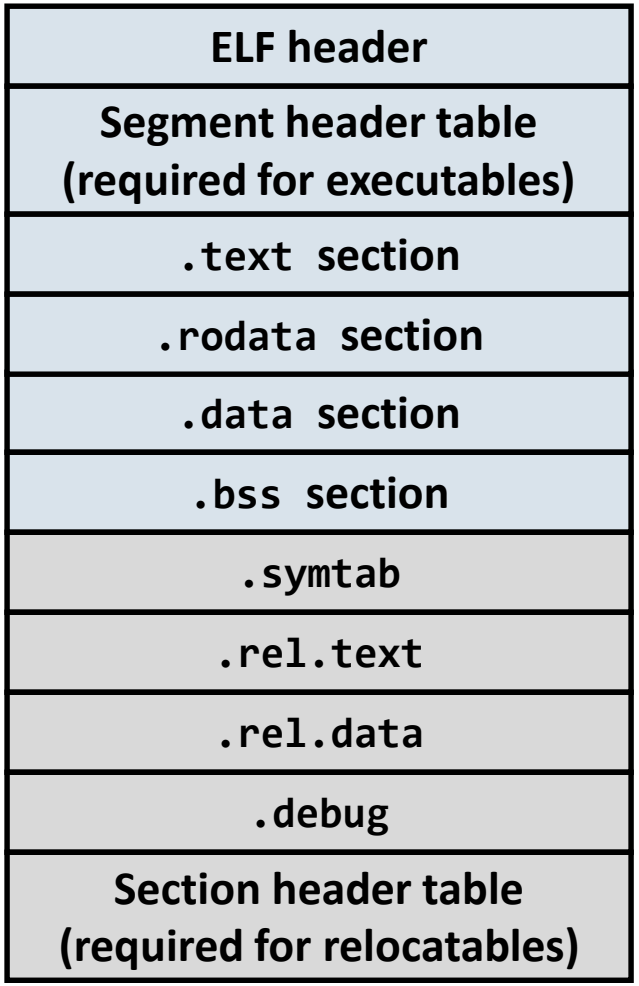
ELF Object File Format (1)

- ELF header
  - Magic number (`\177ELF`),
  - Type (`.o`, `exec`, `.so`)
  - Machine type
  - Word size
  - Byte ordering, etc.
- Segment header table
  - Page size
  - Virtual addresses for memory segments (sections)
  - Segment sizes



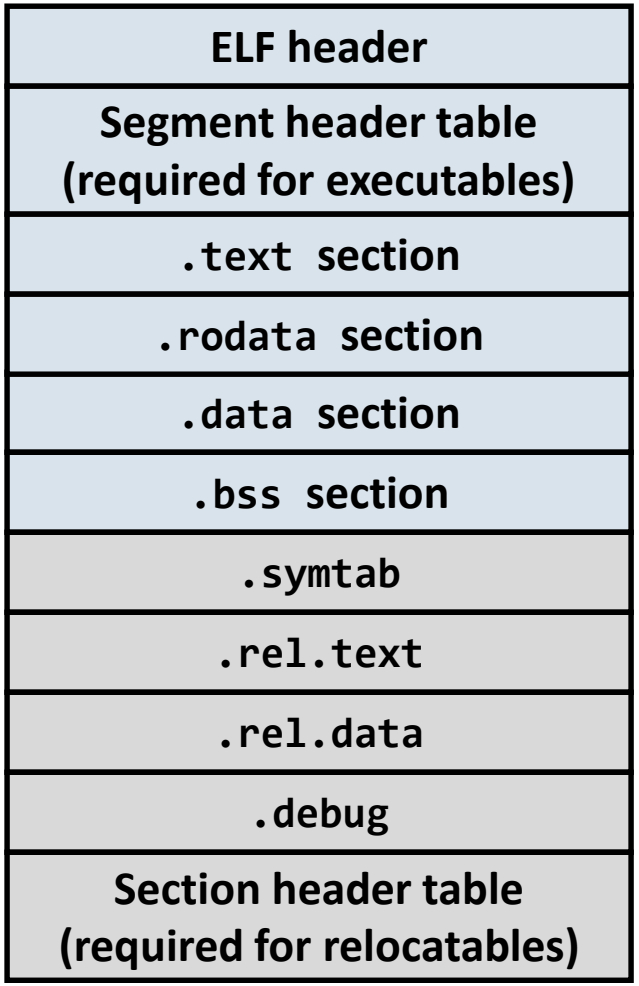
ELF Object File Format (2)

- `.text` section
  - Code
- `.rodata` section
  - Read-only data: strings, etc.
- `.data` section
  - Initialized global variables
- `.bss` section
  - Uninitialized global variables
    - "Block Started by Symbol" or
    - "Better Save Space"
  - Occupies no space



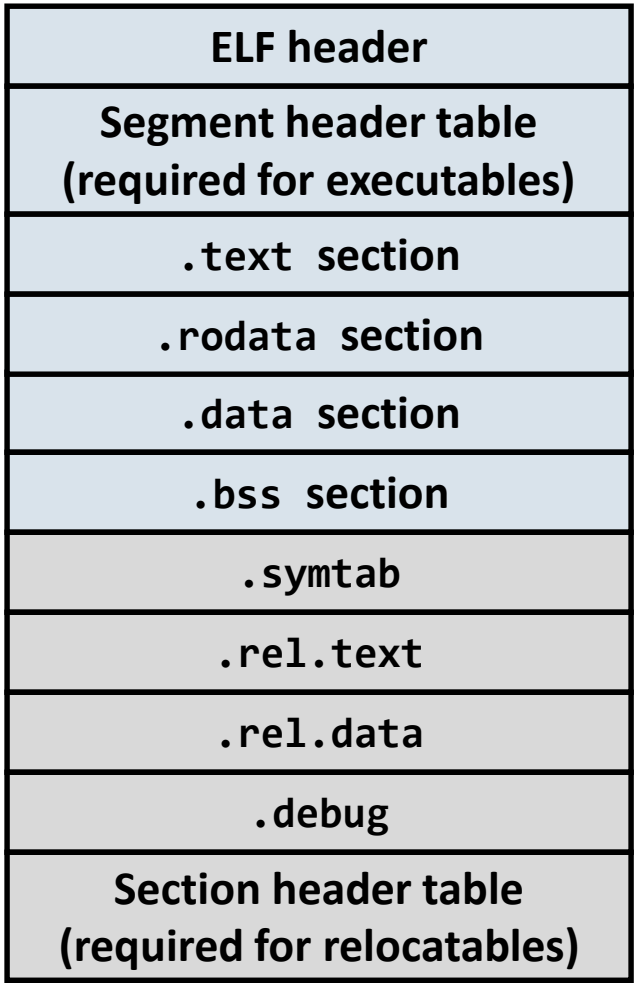
ELF Object File Format (3)

- **`.symtab` section**
  - Symbol table
  - Procedure and static variable names
  - Section names and locations
- **`.rel.text` section**
  - Relocation info for `.text` section
  - Addresses of instructions that will need to be modified in the executable
  - Instructions for modifying

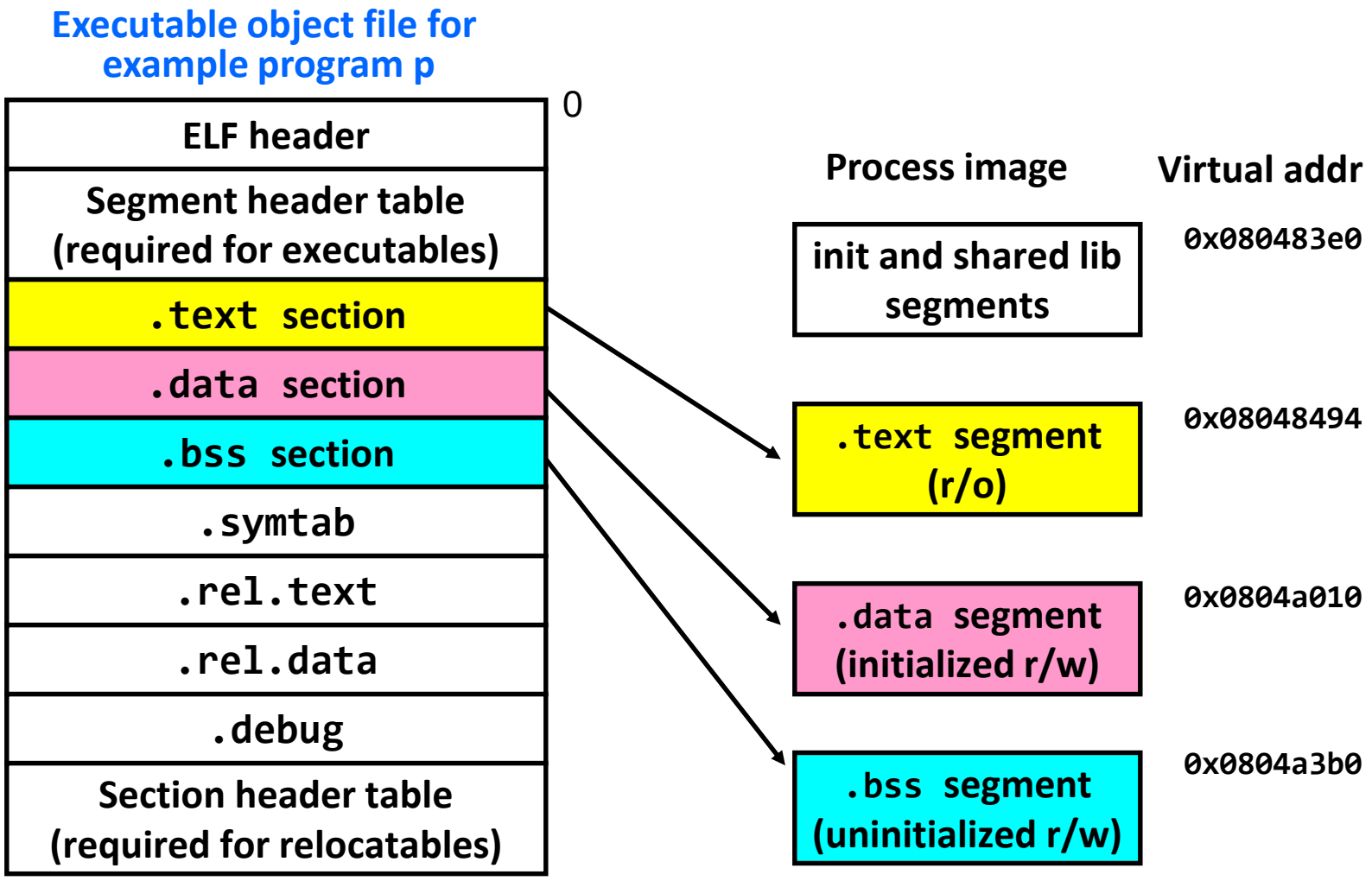


ELF Object File Format (4)

- **`.rel.data` section**
  - Relocation info for **`.data`** section
  - Addresses of pointer data that will need to be modified in the merged executable
- **`.debug` section**
  - Info for symbolic debugging (**`gcc -g`**)
- Section header table
  - Offsets and sizes of each section



Loading Executable Binaries



Linking Example (1)

- Example C program

m.c

```
int e=7;

int main() {
  int r = a();
  exit(0);
}
```

a.c

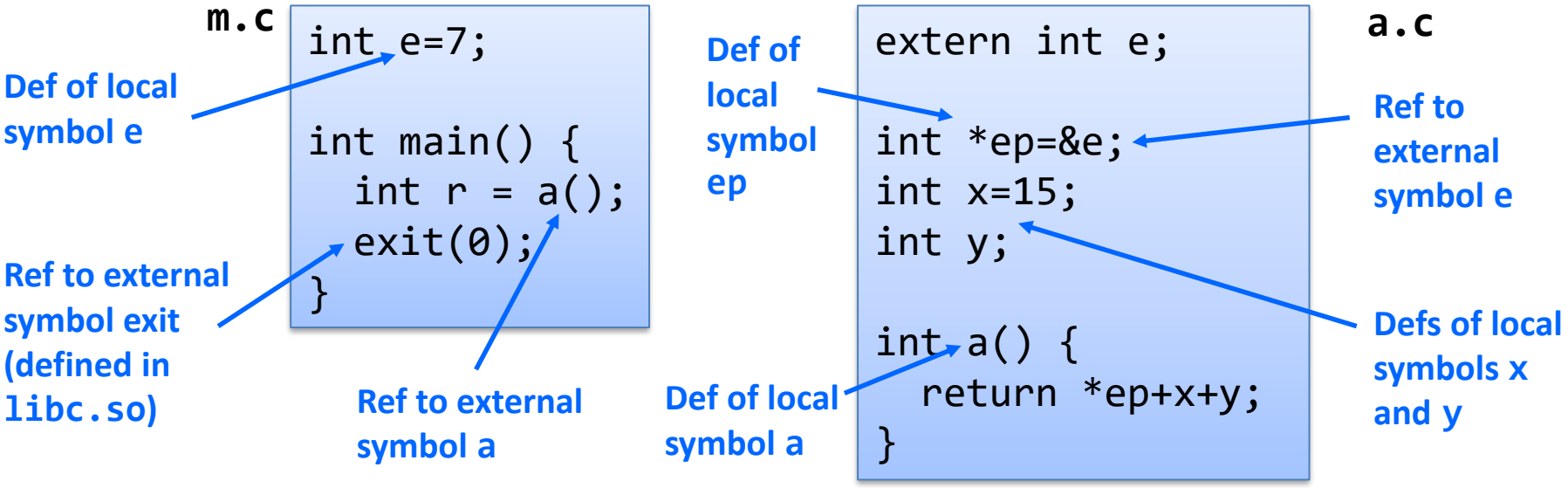
```
extern int e;

int *ep=&e;
int x=15;
int y;

int a() {
  return *ep+x+y;
}
```

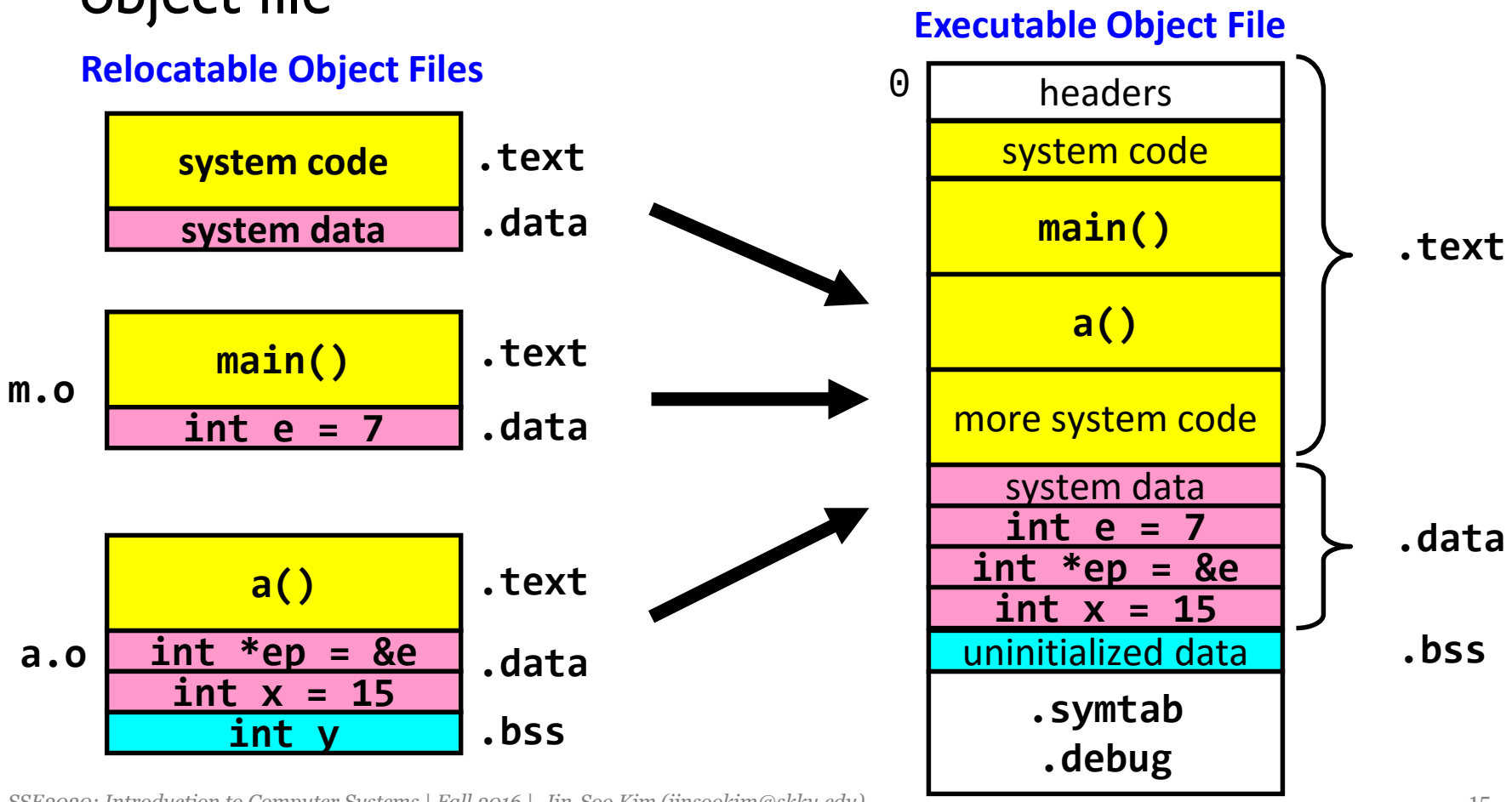
Linking Example (2)

- Relocating symbols and resolving external references
  - Symbols are lexical entities that name functions and variables
  - Each symbol has a value (typically a memory address)
  - Code consists of symbol definitions and references
  - References can be either local or external



Linking Example (3)

- Merging relocatable object files into an executable object file



Packaging Common Functions

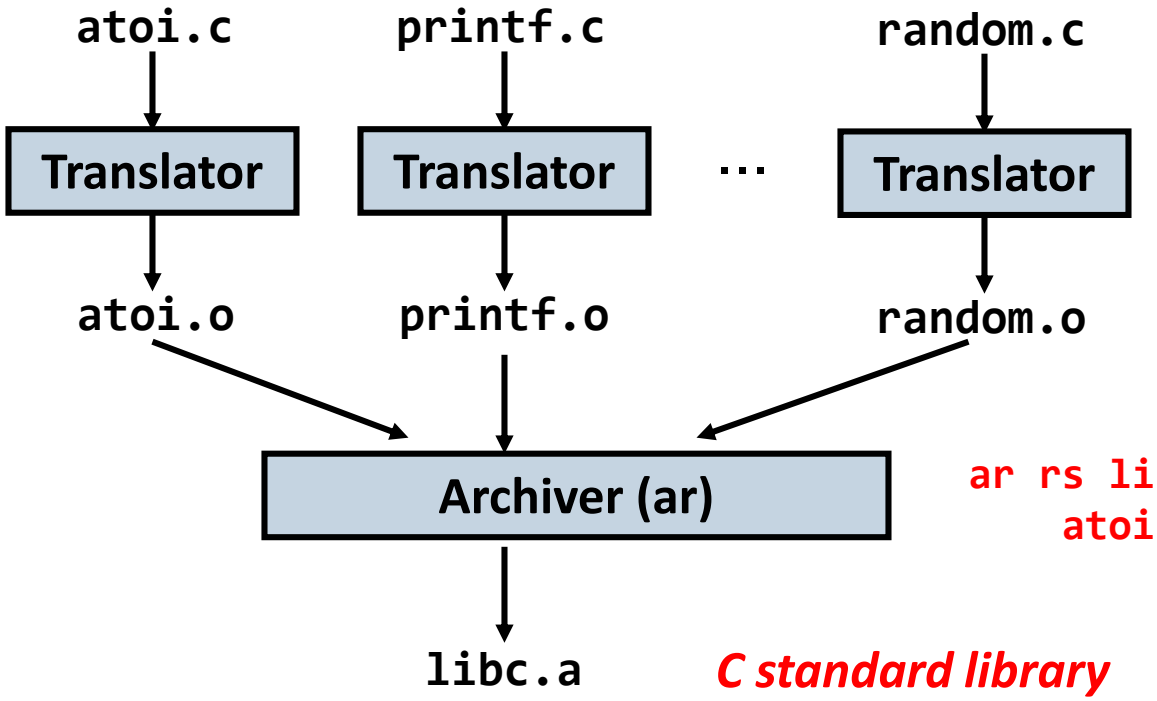
- How to package functions commonly used by programmers?
  - Math, I/O, memory management, string manipulation, etc.
- Option 1: put all functions in a single source file
  - Programmers link big object file into their programs
  - Space and time inefficient
- Option 2: put each function in a separate source file
  - Programmers explicitly link appropriate binaries into their programs
  - More efficient, but burdensome on the programmer

Old-fashioned Solution

- Static libraries (`.a` archive files)
  - Concatenate related relocatable object files into a single file with an index (called an *archive*)
  - Enhance linker so that it tries to resolve unresolved external references by looking or the symbols in one or more archives
  - If an archive member file resolves reference, link it into executable
  - Further improves modularity and efficiency by packaging commonly used functions
    - e.g. C standard library (`libc`), math library (`libm`), etc.

Creating Static Libraries

- Use a tool called archiver (**ar**)
  - Archiver allows incremental updates
  - Recompile functions that changes and replace **.o** file in archive

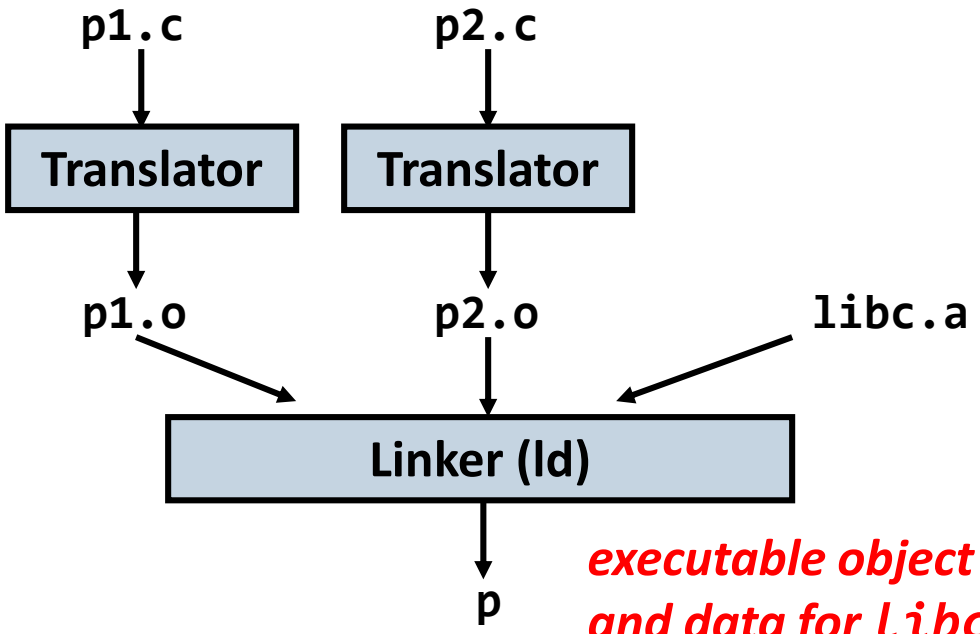



```
ar rs libc.a \
    atoi.o printf.o … random.o
```

C standard library

Linking with Static Libraries

- Link selectively only the **`.o`** files in the archive that are actually needed by the program



static library (archive) of relocatable object files concatenated into one file.

executable object file (only contains code and data for `libc` functions that are called from `p1.c` and `p2.c`)

Commonly Used Libraries

- **`libc.a`** (the C standard library)
  - 4.6MB archive of 1496 object files
  - I/O, memory allocation, signal handling, string handling, data and time, random numbers, integer math
- **`libm.a`** (the C math library)
  - 2MB of 444 object files
  - floating point math (sin, cos, tan, log, exp, sqrt, …)

```
% ar -t libc.a | sort
…
fork.o
fprintf.o
fputc.o
fscanf.o
…
```

```
% ar -t libm.a | sort
…
e_acos.o
e_acosf.o
e_acosh.o
e_acoshf.o
…
```

Problems in Static Libraries

- Potential for duplicating lots of common code in the executable files on a filesystem
  - e.g. Every C program needs the standard C library
- Potential for duplicating lots of code in the virtual memory space of many processes
- Minor bug fixes of system libraries require each application to explicitly relink

Example

@ sys

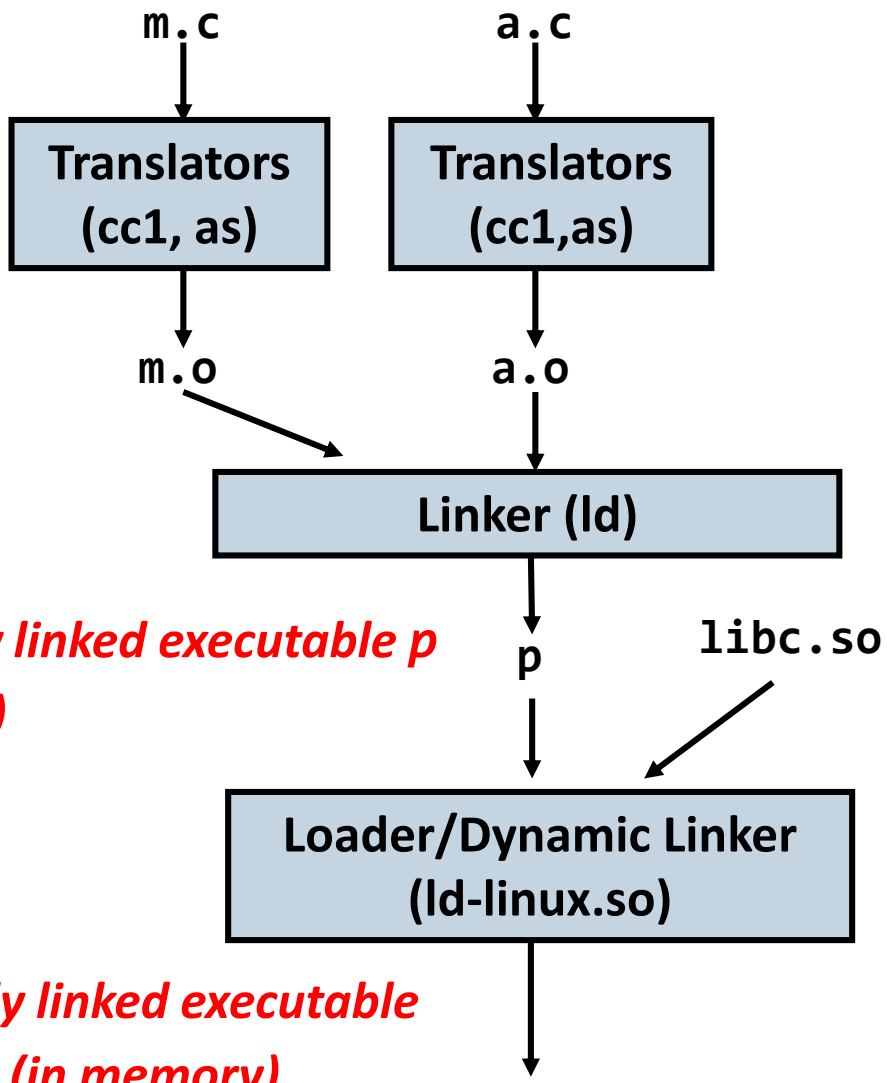
```
[sys:~/tmp-2055] cat hello.c
#include <stdio.h>

int main (void)
{
        printf ("hello, world\n");
}
[sys:~/tmp-2056] gcc -o hello-static -static hello.c
[sys:~/tmp-2057] gcc -o hello-dynamic hello.c
[sys:~/tmp-2058] ls -l hello-*
-rwxrwxr-x 1 jinsoo jinsoo   8600 11월 28 16:10 hello-dynamic
-rwxrwxr-x 1 jinsoo jinsoo 908608 11월 28 16:10 hello-static
[sys:~/tmp-2059] ./hello-static
hello, world
[sys:~/tmp-2060] ./hello-dynamic
hello, world
[sys:~/tmp-2061]
```

Modern Solution: Shared Libraries

- Also called dynamic link libraries or DLLs
  - Object files that contain code and data are loaded and linked into an application dynamically, at either load-time or run-time
- Dynamic linking at load-time
  - Handled automatically by the dynamic linker (`ld-linux.so`)
  - Standard C library (`libc.so`) usually dynamically linked
- Dynamic linking at run-time
  - Done explicitly by user with `dlopen()`
- Shared library routines can be shared by multiple processes

Linking with Shared Libraries



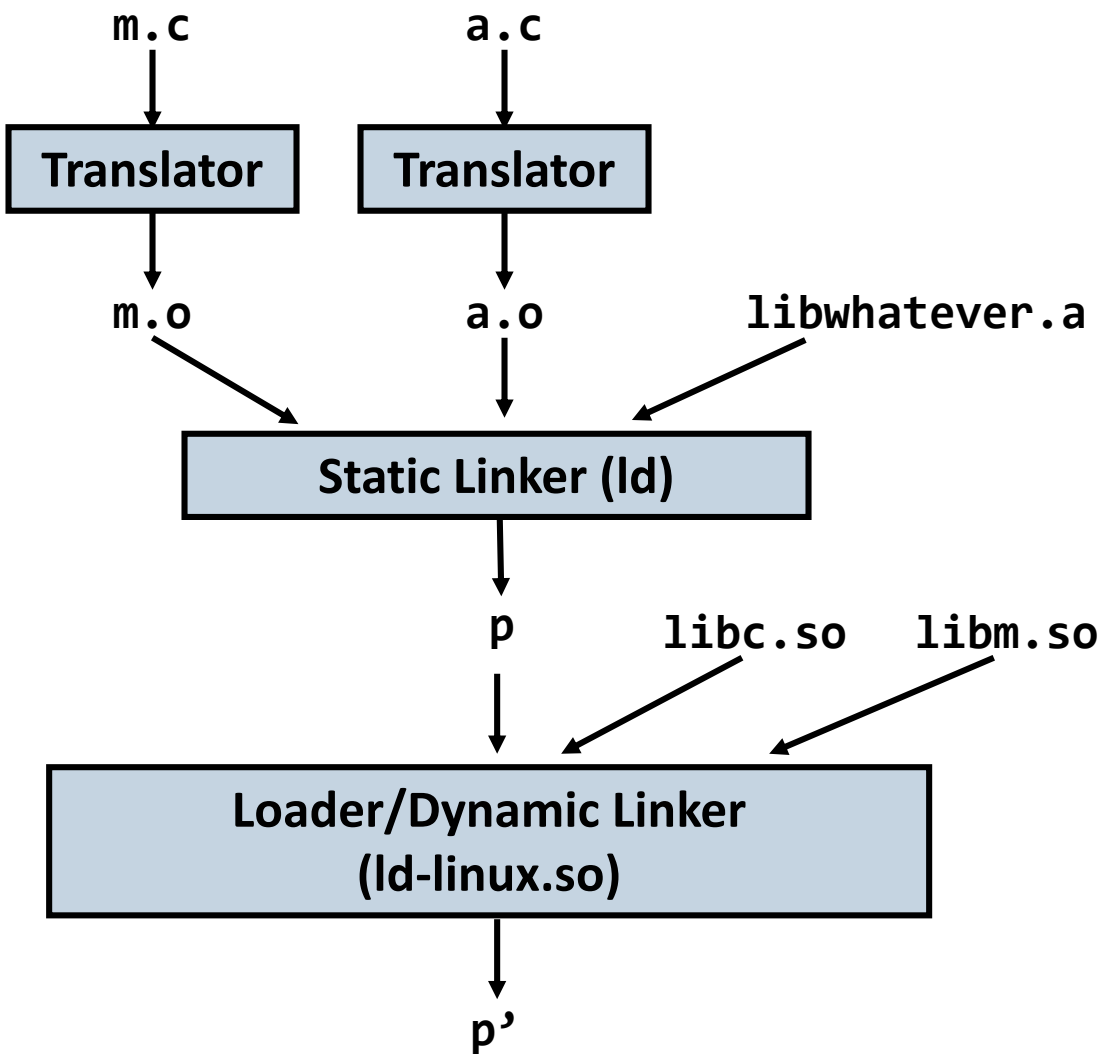
Partially linked executable p (on disk)

Shared library of dynamically relocatable object files

Fully linked executable p' (in memory)

`libc.so` functions called by `m.c` and `a.c` are loaded, linked, and (potentially) shared among processes.

The Complete Picture



Summary

- Linking is a technique that allows programs to be constructed from multiple object files
- Linking can happen at different times in a program's lifetime
  - Compile time (when a program is compiled) – static linking
  - Load time (when a program is loaded into memory)
  - Run time (when a program is executing)
- Understanding linking can help you avoid nasty errors and make you a better programmer