IF2130 – Organisasi dan Arsitektur Komputer

sumber: Greg Kesden, CMU 15-213, 2012

Linking

Achmad Imam Kistijantoro (imam@informatika.org)

Today

- Linking
- Case study: Library interpositioning

Example C Program

main.c

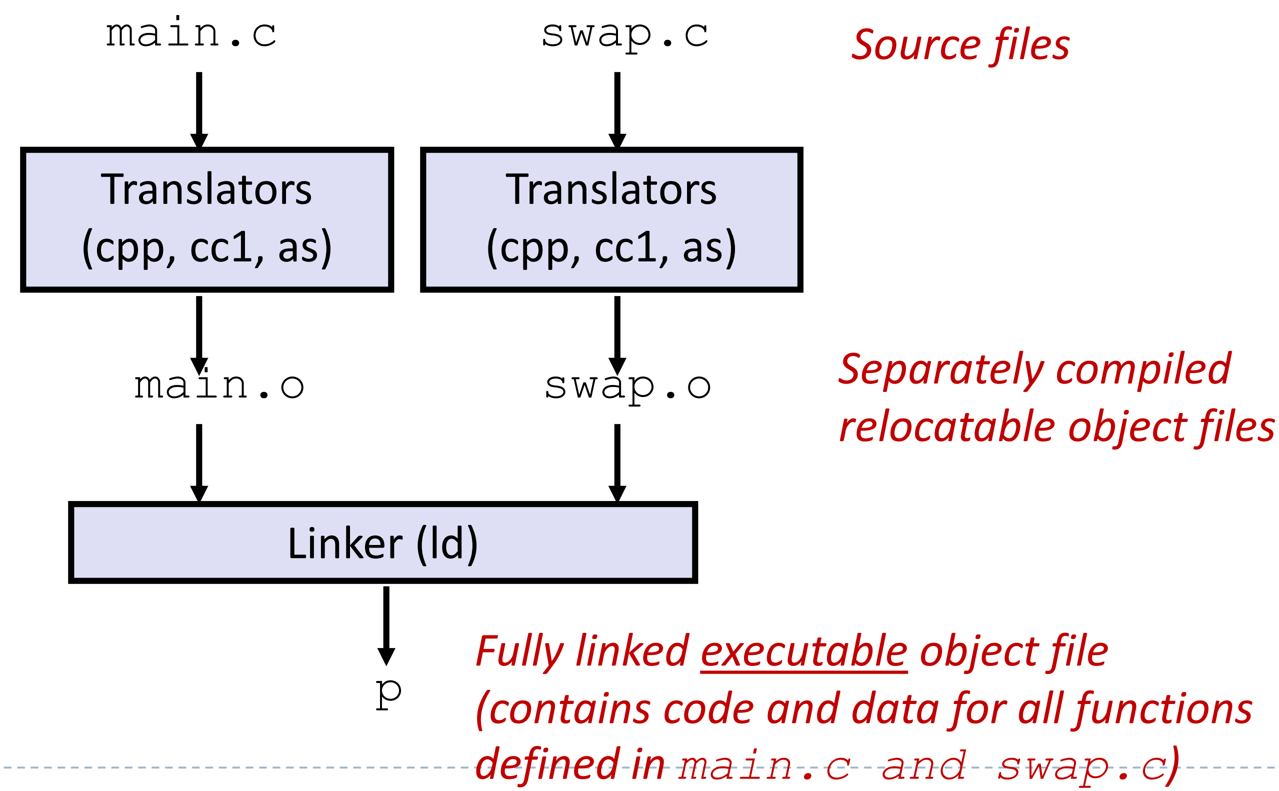
```
int buf[2] = {1, 2};

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

swap.c

```
extern int buf[];

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

void swap()
{
  int temp;

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

Static Linking

- Programs are translated and linked using a *compiler driver*:
  - `unix> gcc -O2 -g -o p main.c swap.c`
  - `unix> ./p`



Why Linkers?

- Reason 1: Modularity

  - Program can be written as a collection of smaller source files, rather than one monolithic mass.

  - Can build libraries of common functions (more on this later)
    - e.g., Math library, standard C library

Why Linkers? (cont)

- Reason 2: Efficiency
  - Time: Separate compilation
    - Change one source file, compile, and then relink.
    - No need to recompile other source files.
  - Space: Libraries
    - Common functions can be aggregated into a single file...
    - Yet executable files and running memory images contain only code for the functions they actually use.

What Do Linkers Do?

- Step 1. Symbol resolution

  - Programs define and reference *symbols* (variables and functions):
    - `void swap() {…}     /* define symbol swap */`
    - `swap();             /* reference symbol a */`
    - `int *xp = &x;       /* define symbol xp, reference x */`

  - Symbol definitions are stored (by compiler) in *symbol table*.
    - Symbol table is an array of structs
    - Each entry includes name, size, and location of symbol.

  - **During symbol resolution step, the linker associates each symbol reference with exactly one symbol definition.**

What Do Linkers Do? (cont)

- Step 2. Relocation
  - Merges separate code and data sections into single sections
  - Relocates symbols from their relative locations in the `.o` files to their final absolute memory locations in the executable.
  - Updates all references to these symbols to reflect their new positions.

Three Kinds of Object Files (Modules)

- Relocatable object file (`.o` file)
  - Contains code and data in a form that can be combined with other relocatable object files to form executable object file.
    - Each `.o` file is produced from exactly one source (`.c`) file
- Executable object file (`a.out` file)
  - Contains code and data in a form that can be copied directly into memory and then executed.
- Shared object file (`.so` file)
  - 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) by Windows

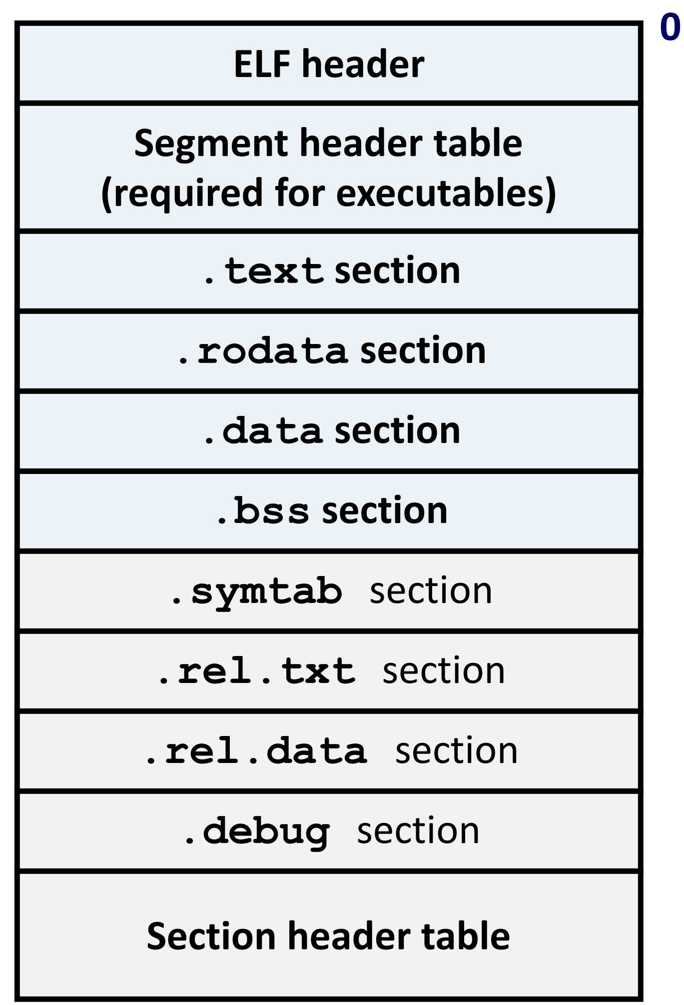
Executable and Linkable Format (ELF)

- Standard binary format for object files
- Originally proposed by AT&T System V Unix
  - Later adopted by BSD Unix variants and Linux
- One unified format for
  - Relocatable object files (`.o`),
  - Executable object files (`a.out`)
  - Shared object files (`.so`)

- Generic name: ELF binaries

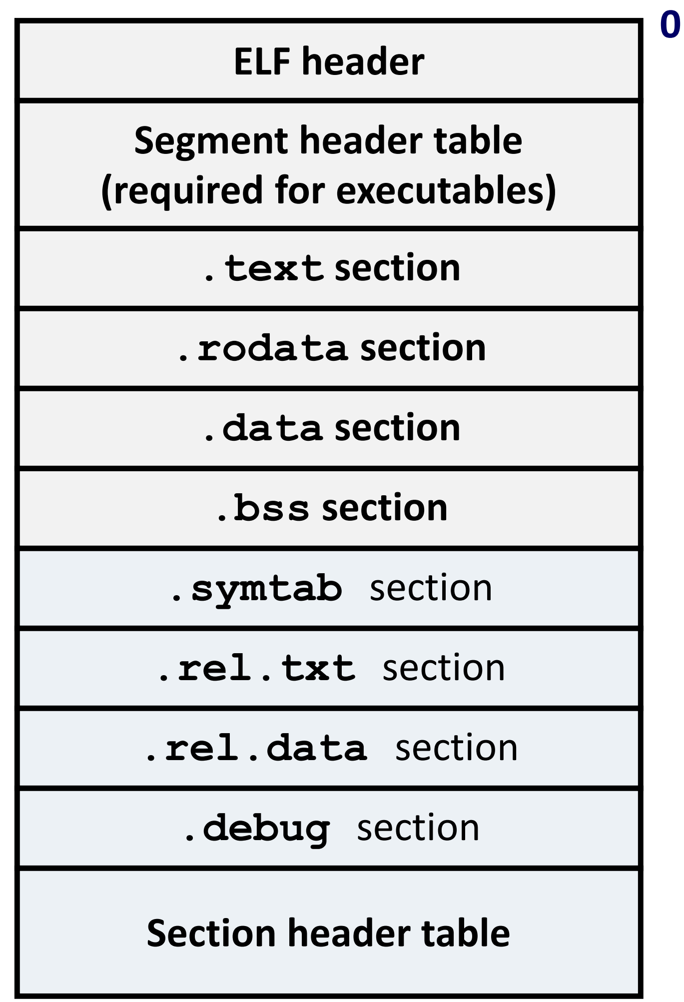
ELF Object File Format

- Elf header
  - Word size, byte ordering, file type (.o, exec, .so), machine type, etc.
- Segment header table
  - Page size, virtual addresses memory segments (sections), segment sizes.
- `.text` section
  - Code
- `.rodata` section
  - Read only data: jump tables, ...
- `.data` section
  - Initialized global variables
- `.bss` section
  - Uninitialized global variables
  - "Block Started by Symbol"
  - "Better Save Space"
  - Has section header but occupies no space



ELF Object File Format (cont.)

- `.symtab` **section**
  - Symbol table
  - Procedure and global non-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.
- `.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



Linker Symbols

- Global symbols
  - Symbols defined by module *m* that can be referenced by other modules.
  - E.g.: non-`static` C functions and non-`static` global variables.
- External symbols
  - Global symbols that are referenced by module *m* but defined by some other module.
- Local symbols
  - Symbols that are defined and referenced exclusively by module *m*.
  - E.g.: C functions and variables defined with the `static` attribute.
  - **Local linker symbols are *not* local program variables**

Resolving Symbols

Global

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

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

main.c

External

Global External Local

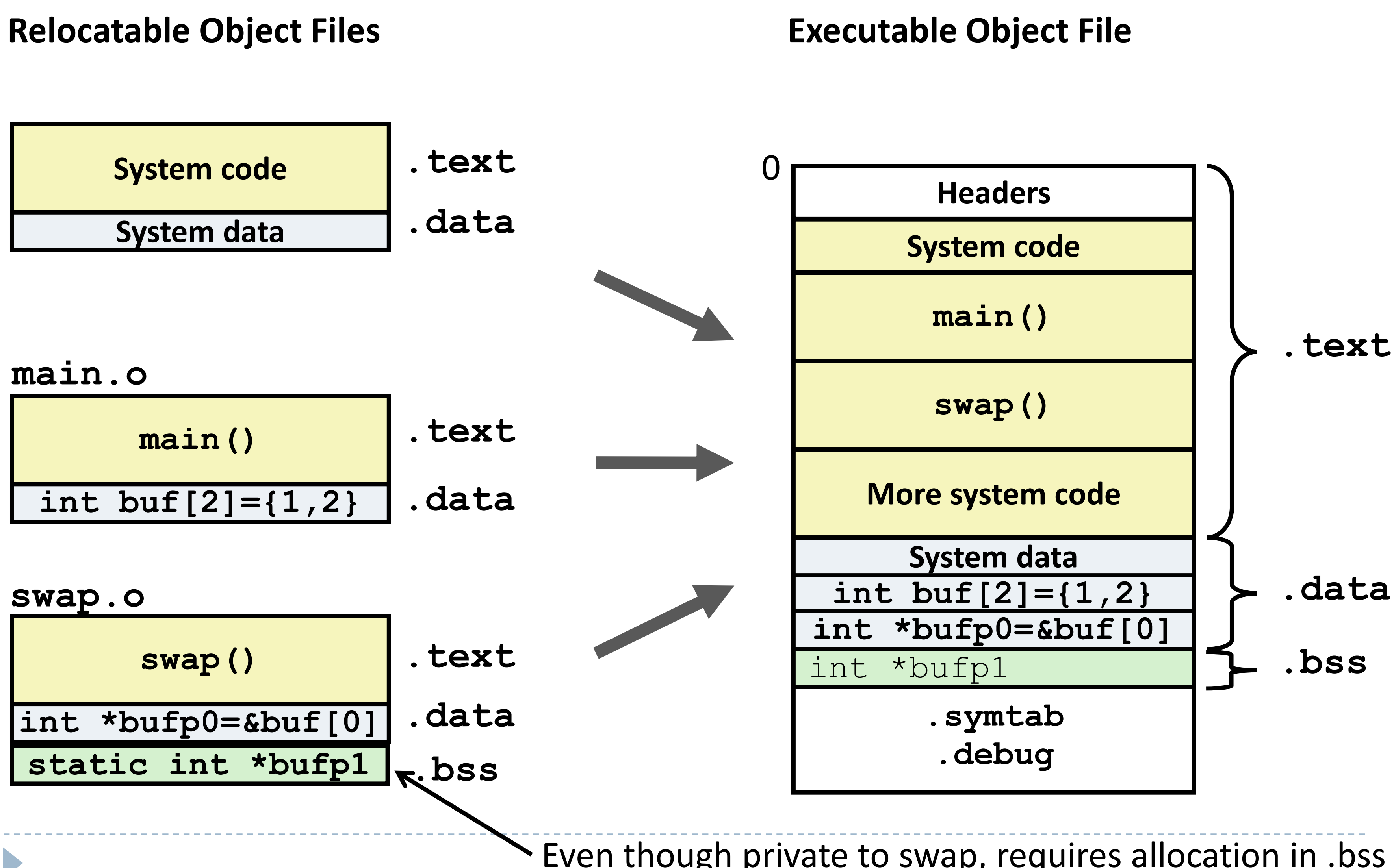
```c
extern int buf[];

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

Relocating Code and Data



2 Relocation types

- R_386_PC32
  - relocate a reference that uses a 32-bit PC-relative address.
  - Effective address = PC + instruction encoded addr
- R_386_32
  - Absolute addressing
  - Uses the value encoded in the instruction

Relocation Info (main)

main.c

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

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

main.o

```
0000000 <main>:
   0:   8d 4c 24 04         lea    0x4(%esp),%ecx
   4:   83 e4 f0            and    $0xfffffff0,%esp
   7:   ff 71 fc            pushl  0xfffffffc(%ecx)
   a:   55                  push   %ebp
   b:   89 e5               mov    %esp,%ebp
   d:   51                  push   %ecx
   e:   83 ec 04            sub    $0x4,%esp
  11:   e8 fc ff ff ff      call   12 <main+0x12>
                    12: R_386_PC32 swap
  16:   83 c4 04            add    $0x4,%esp
  19:   31 c0               xor    %eax,%eax
  1b:   59                  pop    %ecx
  1c:   5d                  pop    %ebp
  1d:   8d 61 fc            lea    0xfffffffc(%ecx),%esp
  20:   c3                  ret
```

```
Disassembly of section .data:

 00000000 <buf>:
   0:   01 00 00 00 02 00 00 00
```

Source: `objdump -r -d`

Relocation Info (swap, .text)

swap.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.o

```
Disassembly of section .text:

00000000 <swap>:
   0:   8b 15 00 00 00 00       mov    0x0,%edx
              2: R_386_32       buf
   6:   a1 04 00 00 00          mov    0x4,%eax
              7: R_386_32       buf
   b:   55                      push   %ebp
   c:   89 e5                   mov    %esp,%ebp
   e:   c7 05 00 00 00 00 04    movl   $0x4,0x0
  15:   00 00 00
              10: R_386_32      .bss
              14: R_386_32      buf
  18:   8b 08                   mov    (%eax),%ecx
  1a:   89 10                   mov    %edx,(%eax)
  1c:   5d                      pop    %ebp
  1d:   89 0d 04 00 00 00       mov    %ecx,0x4
              1f: R_386_32      buf
  23:   c3                      ret
```

Relocation Info (swap, `.data`)

swap.c

```
extern int buf[];

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

void swap()
{
  int temp;

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

```
Disassembly of section .data:

00000000 <bufp0>:
   0:    00 00 00 00

         0: R_386_32 buf
```

Executable Before/After Relocation (.text)

```
0000000 <main>:
   . . .
   e:   83 ec 04            sub     $0x4,%esp
  11:   e8 fc ff ff ff      call    12 <main+0x12>
                 12: R_386_PC32 swap
  16:   83 c4 04            add     $0x4,%esp
  . . .
```

0x8048396 + 0x1a
= 0x80483b0

```
08048380 <main>:
 8048380:      8d 4c 24 04             lea    0x4(%esp),%ecx
 8048384:      83 e4 f0                and    $0xfffffff0,%esp
 8048387:      ff 71 fc                pushl  0xfffffffc(%ecx)
 804838a:      55                      push   %ebp
 804838b:      89 e5                   mov    %esp,%ebp
 804838d:      51                      push   %ecx
 804838e:      83 ec 04                sub    $0x4,%esp
 8048391:      e8 1a 00 00 00          call   80483b0 <swap>
 8048396:      83 c4 04                add    $0x4,%esp
 8048399:      31 c0                   xor    %eax,%eax
 804839b:      59                      pop    %ecx
 804839c:      5d                      pop    %ebp
 804839d:      8d 61 fc                lea    0xfffffffc(%ecx),%esp
 80483a0:      c3                      ret
```

```
   0:   8b 15 00 00 00 00       mov     0x0,%edx
              2: R_386_32       buf
   6:   a1 04 00 00 00          mov     0x4,%eax
              7: R_386_32       buf
  ...
   e:   c7 05 00 00 00 00 04    movl    $0x4,0x0
  15:   00 00 00
              10: R_386_32      .bss
              14: R_386_32      buf
  . . .
  1d:   89 0d 04 00 00 00       mov     %ecx,0x4
              1f: R_386_32      buf
  23:   c3                      ret
```

```
080483b0 <swap>:
 80483b0:      8b 15 20 96 04 08       mov    0x8049620,%edx
 80483b6:      a1 24 96 04 08          mov    0x8049624,%eax
 80483bb:      55                      push   %ebp
 80483bc:      89 e5                   mov    %esp,%ebp
 80483be:      c7 05 30 96 04 08 24    movl   $0x8049624,0x8049630
 80483c5:      96 04 08
 80483c8:      8b 08                   mov    (%eax),%ecx
 80483ca:      89 10                   mov    %edx,(%eax)
 80483cc:      5d                      pop    %ebp
 80483cd:      89 0d 24 96 04 08       mov    %ecx,0x8049624
 80483d3:      c3                      ret
```

Executable After Relocation (.data)

```
Disassembly of section .data:

08049620 <buf>:
 8049620:       01 00 00 00 02 00 00 00

08049628 <bufp0>:
 8049628:       20 96 04 08
```

Strong and Weak Symbols

- Program symbols are either strong or weak
  - ***Strong***: procedures and initialized globals
  - ***Weak***: uninitialized globals

p1.c

```
int foo=5;     <- strong

p1() {         <- strong
}
```

p2.c

```
int foo;       <- weak

p2() {         <- strong
}
```

Linker's Symbol Rules

- Rule 1: Multiple strong symbols are not allowed
  - Each item can be defined only once
  - Otherwise: Linker error
- Rule 2: Given a strong symbol and multiple weak symbol, choose the strong symbol
  - References to the weak symbol resolve to the strong symbol
- Rule 3: If there are multiple weak symbols, pick an arbitrary one
  - Can override this with `gcc -fno-common`

Linker Puzzles

`int x;` `p1() {}`	`p1() {}`	Link time error: two strong symbols (`p1`)
`int x;` `p1() {}`	`int x;` `p2() {}`	References to `x` will refer to the same uninitialized int. Is this what you really want?

`int x;`
`int y;`
`p1() {}`
	`double x;`
	`p2() {}`
		Writes to `x` in `p2` might overwrite `y`! Evil!

`int x=7;`
`int y=5;`
`p1() {}`
	`double x;`
	`p2() {}`
		Writes to `x` in `p2` will overwrite `y`! Nasty!

`int x=7;` `p1() {}`	`int x;` `p2() {}`	References to `x` will refer to the same initialized variable.

Nightmare scenario: two identical weak structs, compiled by different compilers with different alignment rules.

Role of .h Files

c1.c

```c
#include "global.h"

int f() {
  return g+1;
}
```

global.h

```c
#ifdef INITIALIZE
int g = 23;
static int init = 1;
#else
int g;
static int init = 0;
#endif
```

c2.c

```c
#include <stdio.h>
#include "global.h"

int main() {
  if (!init)
    g = 37;
  int t = f();
  printf("Calling f yields %d\n", t);
  return 0;
}
```

Running Preprocessor

c1.c

```c
#include "global.h"

int f() {
  return g+1;
}
```

global.h

```c
#ifdef INITIALIZE
int g = 23;
static int init = 1;
#else
int g;
static int init = 0;
#endif
```

-DINITIALIZE

```c
int g = 23;
static int init = 1;
int f() {
  return g+1;
}
```

no initialization

```c
int g;
static int init = 0;
int f() {
  return g+1;
}
```

- #include causes C preprocessor to insert file verbatim

Role of .h Files (cont.)

global.h

```
#ifdef INITIALIZE
int g = 23;
static int init = 1;
#else
int g;
static int init = 0;
#endif
```

c1.c

```
#include "global.h"

int f() {
  return g+1;
}
```

c2.c

```
#include <stdio.h>
#include "global.h"

int main() {
  if (!init)
    g = 37;
  int t = f();
  printf("Calling f yields %d\n", t);
  return 0;
}
```

What happens:

```
gcc -o p c1.c c2.c
```

??

```
gcc -o p c1.c c2.c \
  -DINITIALIZE
```

??

Global Variables

- Avoid if you can

- Otherwise
  - Use **`static`** if you can
  - Initialize if you define a global variable
  - Use **`extern`** if you use external global variable

Packaging Commonly Used Functions

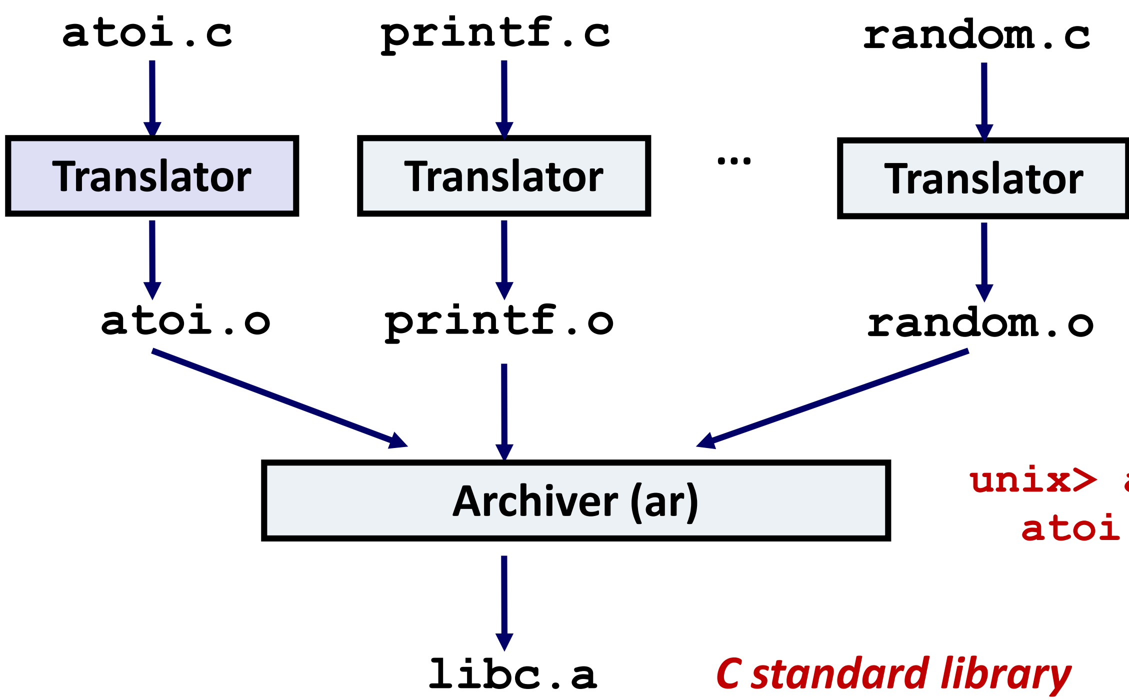
- How to package functions commonly used by programmers?
  - Math, I/O, memory management, string manipulation, etc.

- Awkward, given the linker framework so far:
  - **Option 1:** Put all functions into 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

Solution: Static Libraries

- 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 for the symbols in one or more archives.
  - If an archive member file resolves reference, link it into the executable.

Creating Static Libraries




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

- Archiver allows incremental updates
- Recompile function that changes and replace .o file in archive.

Commonly Used Libraries

`libc.a` (the C standard library)

- 8 MB archive of 1392 object files.
- I/O, memory allocation, signal handling, string handling, data and time, random numbers, integer math

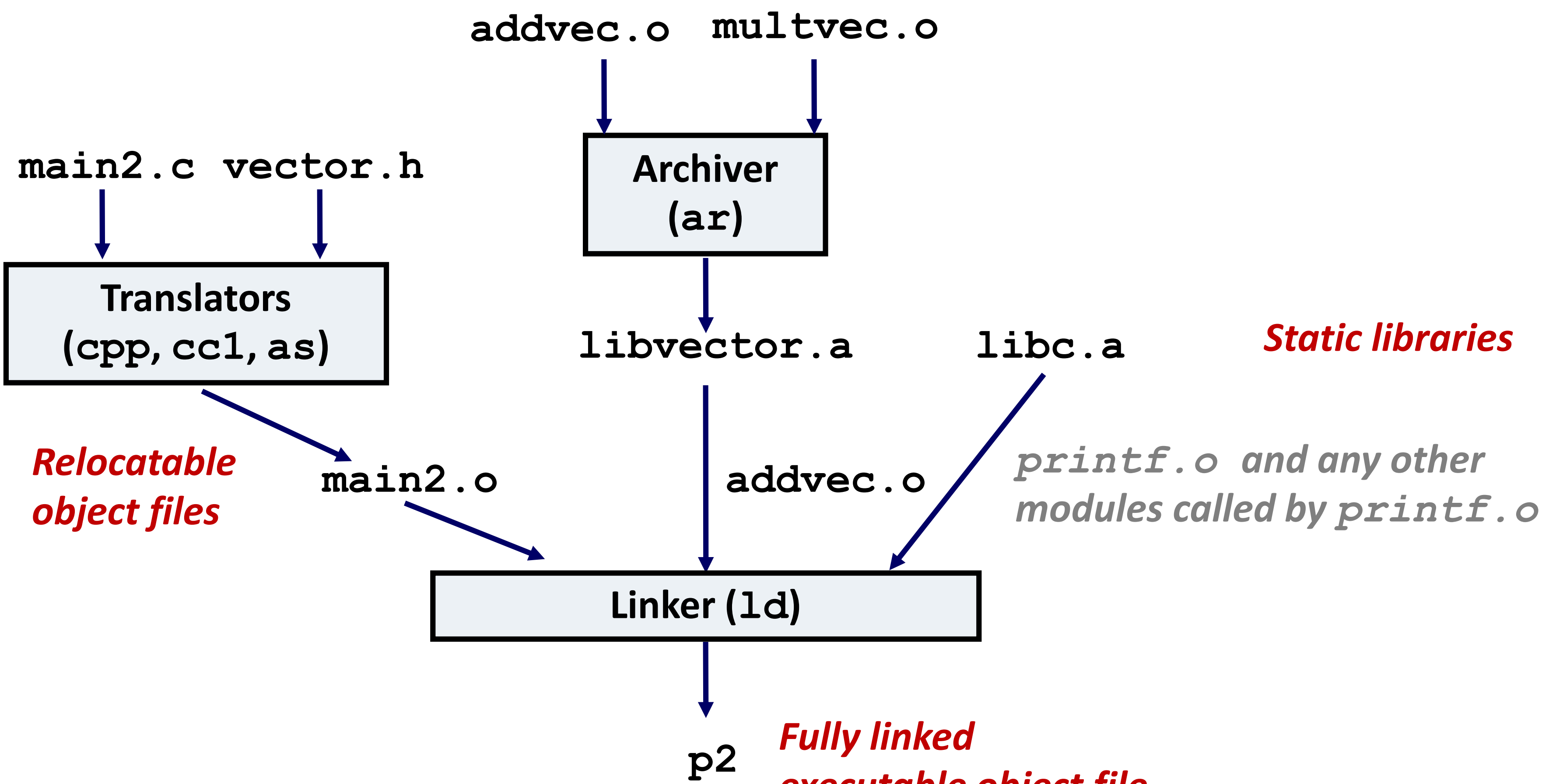
`libm.a` (the C math library)

- 1 MB archive of 401 object files.
- floating point math (sin, cos, tan, log, exp, sqrt, …)

```
% ar -t /usr/lib/libc.a | sort
…
fork.o
…
fprintf.o
fpu_control.o
fputc.o
freopen.o
fscanf.o
fseek.o
fstab.o
…
```

```
% ar -t /usr/lib/libm.a | sort
…
e_acos.o
e_acosf.o
e_acosh.o
e_acoshf.o
e_acoshl.o
e_acosl.o
e_asin.o
e_asinf.o
e_asinl.o
…
```

Linking with Static Libraries

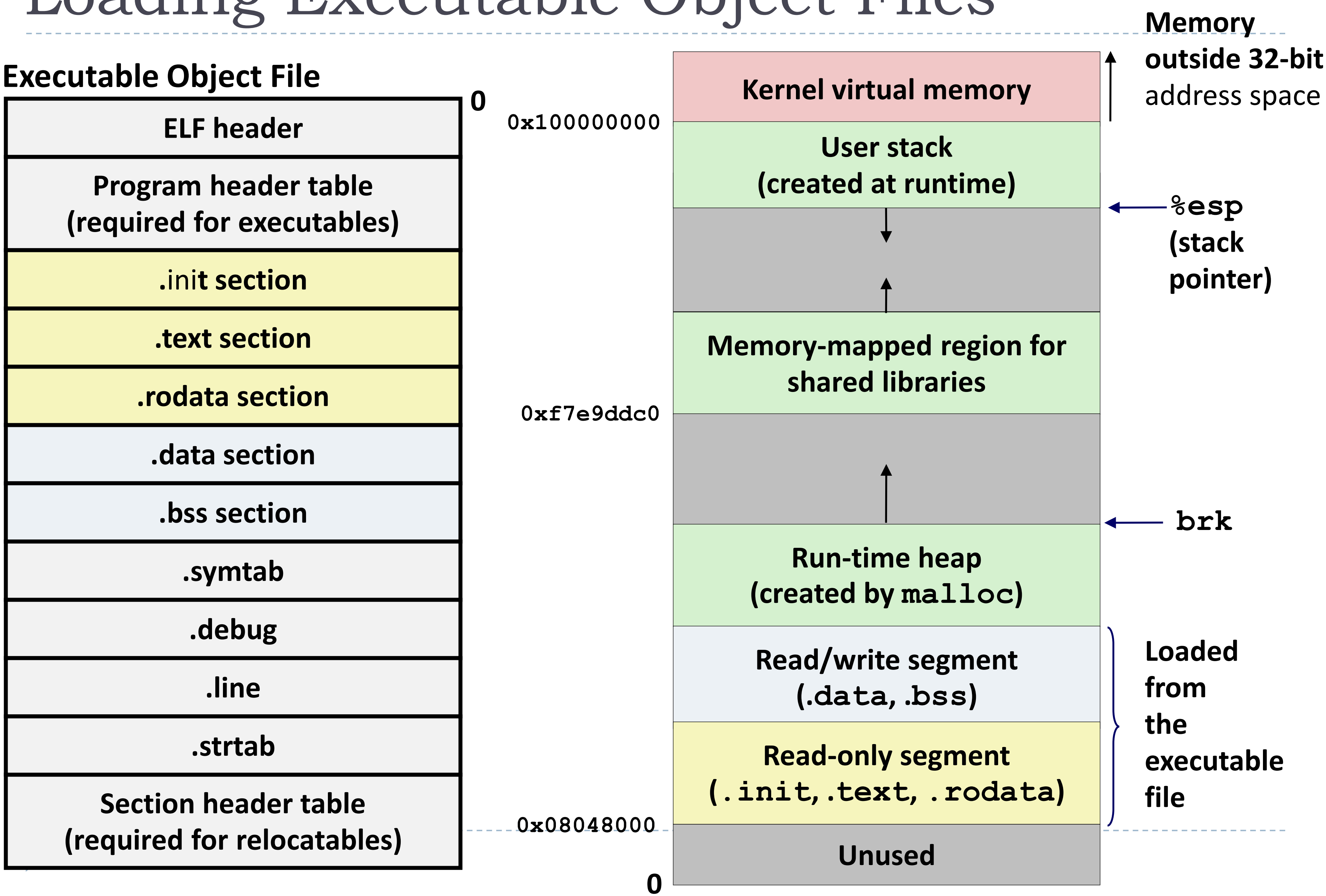


Using Static Libraries

- 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!
  - ```
    unix> gcc -L. libtest.o -lmine
    unix> gcc -L. -lmine libtest.o
    libtest.o: In function `main':
    libtest.o(.text+0x4): undefined reference to `libfun'
    ```
    ```

Loading Executable Object Files



Shared Libraries

- Static libraries have the following disadvantages:
  - Duplication in the stored executables (every function need std libc)
  - Duplication in the running executables
  - Minor bug fixes of system libraries require each application to explicitly relink

- 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*
  - Also called: dynamic link libraries, DLLs, `.so` files

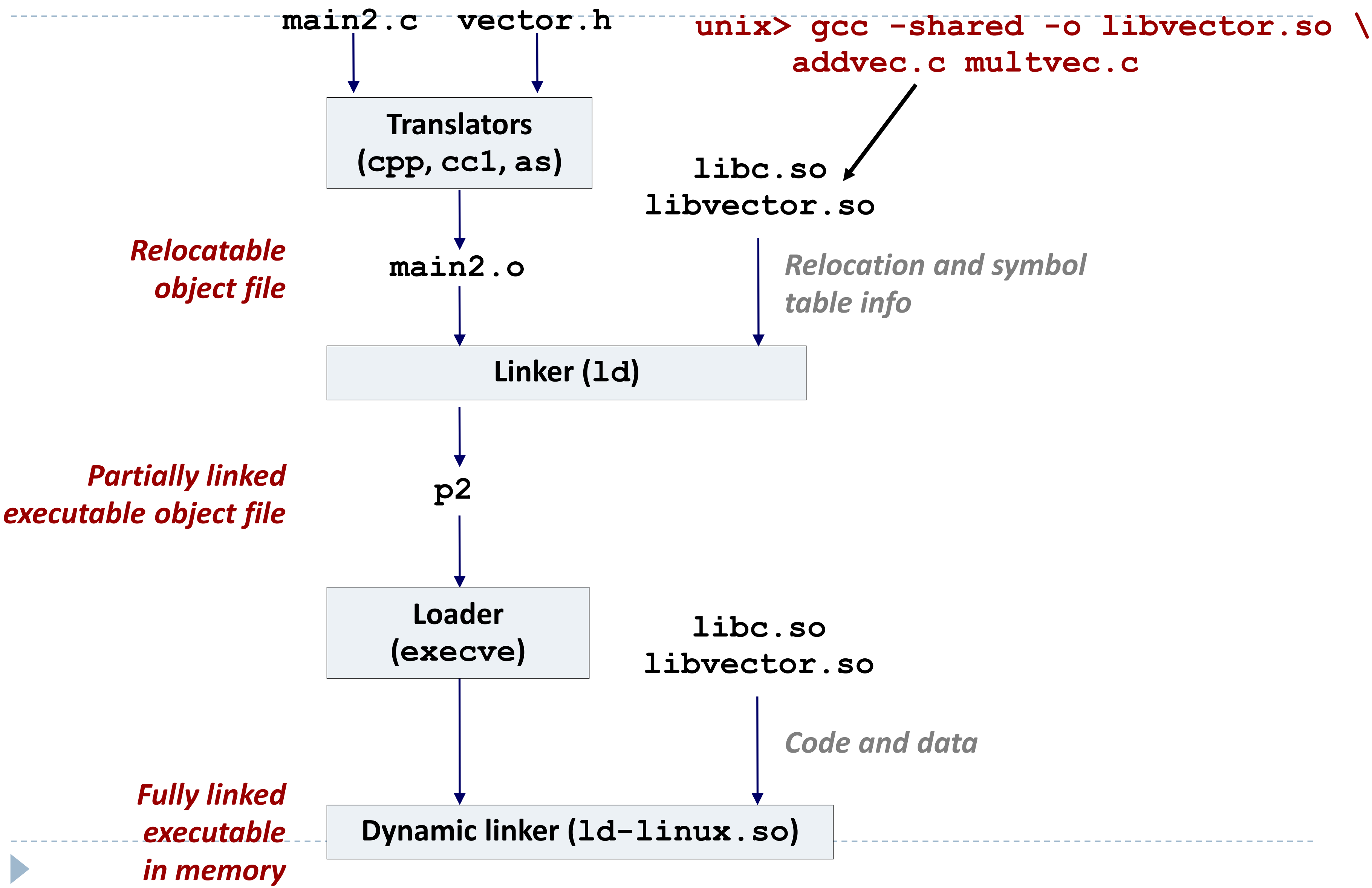
Shared Libraries (cont.)

- Dynamic linking can occur when executable is first loaded and run (load-time linking).
  - Common case for Linux, handled automatically by the dynamic linker (`ld-linux.so`).
  - Standard C library (`libc.so`) usually dynamically linked.

- Dynamic linking can also occur after program has begun (run-time linking).
  - In Linux, this is done by calls to the `dlopen()` interface.
    - Distributing software.
    - High-performance web servers.

- Shared library routines can be shared by multiple processes.
  - More on this when we learn about virtual memory

Dynamic Linking at Load-time



Dynamic Linking at Run-time

```
#include <stdio.h>
#include <dlfcn.h>

int x[2] = {1, 2};
int y[2] = {3, 4};
int z[2];

int main()
{
    void *handle;
    void (*addvec)(int *, int *, int *, int);
    char *error;

    /* dynamically load the shared lib that contains addvec() */
    handle = dlopen("./libvector.so", RTLD_LAZY);
    if (!handle) {
        fprintf(stderr, "%s\n", dlerror());
        exit(1);
    }
```

Dynamic Linking at Run-time

```
    ...

    /* get a pointer to the addvec() function we just loaded */
    addvec = dlsym(handle, "addvec");
    if ((error = dlerror()) != NULL) {
        fprintf(stderr, "%s\n", error);
        exit(1);
    }

    /* Now we can call addvec() just like any other function */
    addvec(x, y, z, 2);
    printf("z = [%d %d]\n", z[0], z[1]);

    /* unload the shared library */
    if (dlclose(handle) < 0) {
        fprintf(stderr, "%s\n", dlerror());
        exit(1);
    }
    return 0;
}
```

Additional Notes

DLL

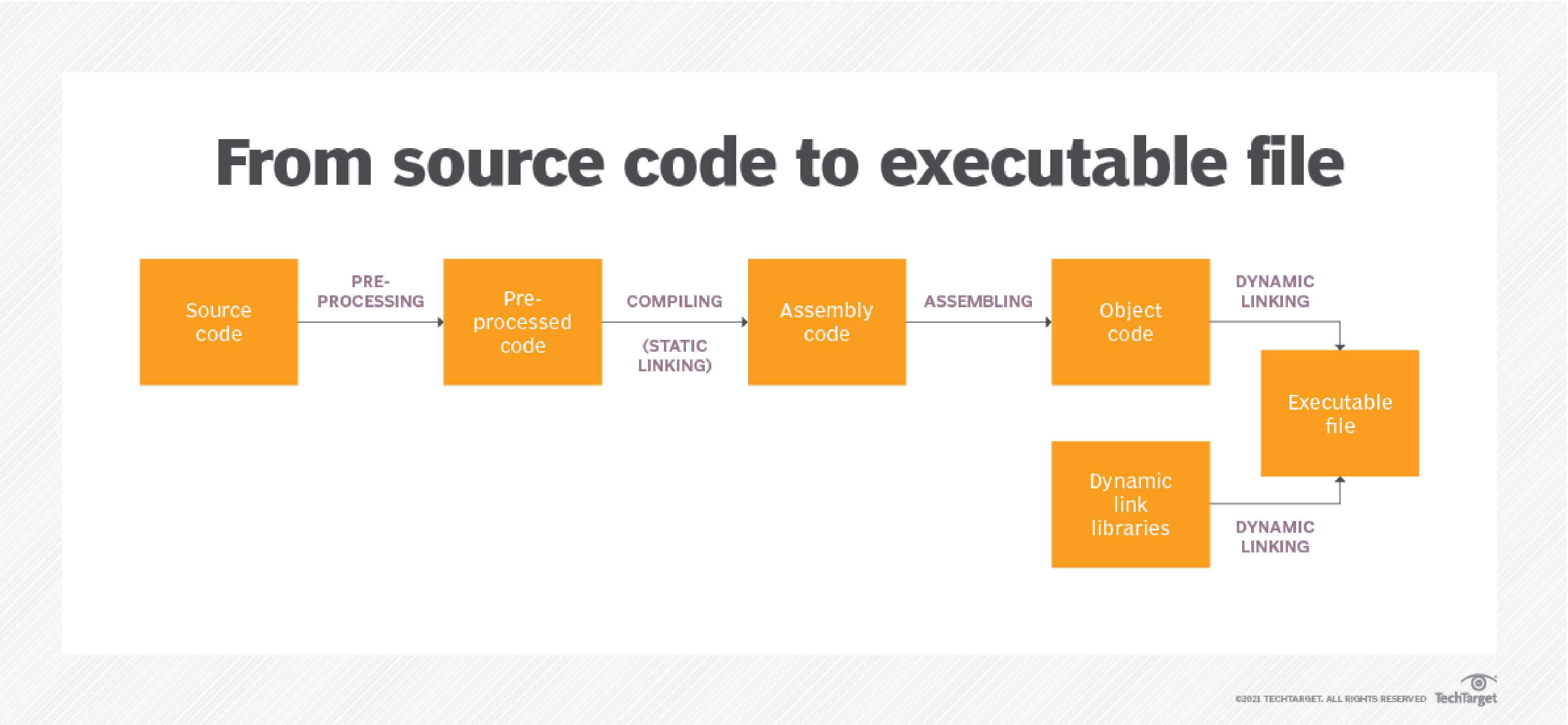
DLL

- A dynamic link library (DLL) is a collection of small programs that larger programs can load when needed to complete specific tasks. The small program, called a DLL file, contains instructions that help the larger program handle what may not be a core function of the original program
- DLL contains bits of code and data, like classes and variables, or other resources such as images that the larger program can use

How does a dynamic link library work?

- Computer programs are rarely written in a one file. They often are composed of multiple files that are linked together.
- When a program is run, it must be compiled from its source code, which is human readable code that the programmer writes. It's turned into an executable file, which is binary code, or machine code, that the computer can read.

- The computer goes through several intermediate steps for this to occur. During those steps, multiple files are linked to one

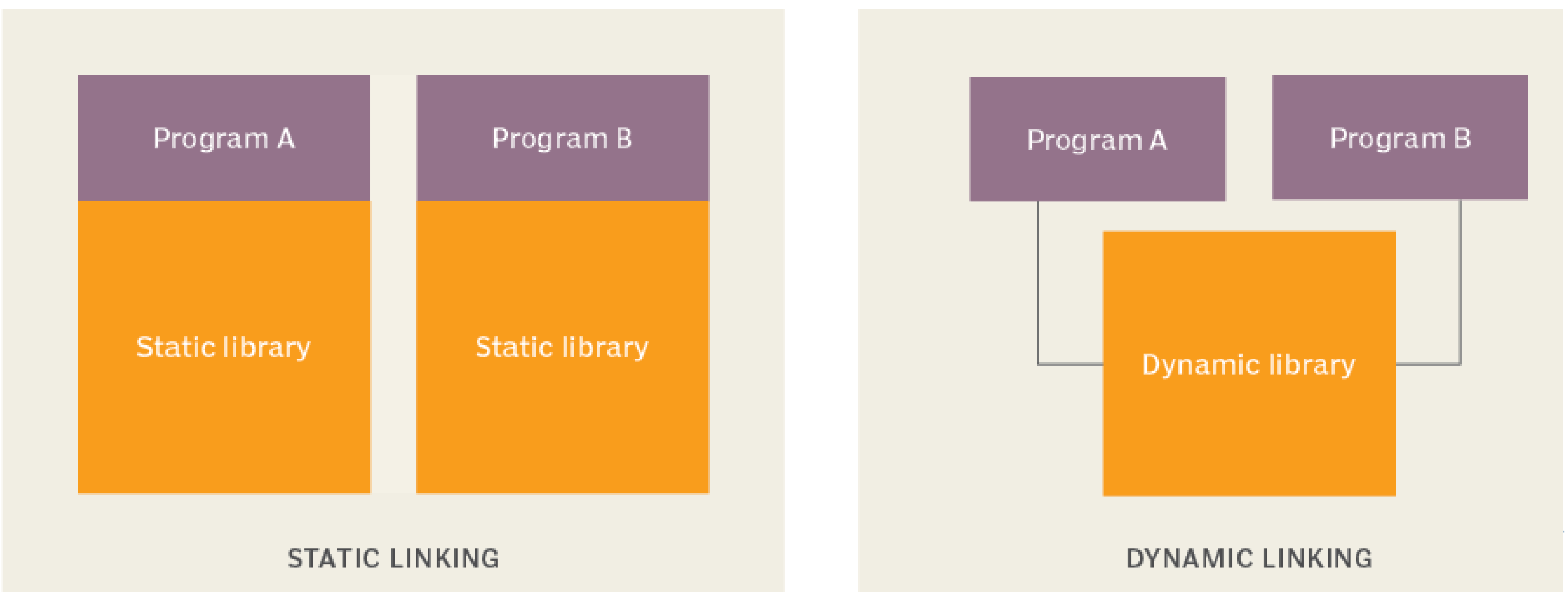


Two Types of Linking

- **Static links.** These are linked earlier in the process and are embedded into the executable. Static libraries are linked to the executable when the program is compiled. Dynamic libraries are linked later, either at runtime or at load time. Static libraries are not shared between programs because they are written into the individual executable.
- **Dynamic links.** DLLs contain the files that a program links to. The libraries already are stored on the computer, external to the program that the user writes. They are called dynamic because they are not embedded in the executable -- they just link to it when needed.

- More than one application can access a dynamic library at once, because they are not embedded with the executable at compile time. Static libraries are embedded into programs, which lead to duplicates among the multiple programs using them.

Static linking vs. dynamic linking



- A dynamically linked program has a small bit of code that maps the DLL into virtual memory, where the program can access it at runtime or load time. With this setup, the dynamically linked program doesn't have to repeatedly access physical memory to access the library. Virtual memory links the same page of physical memory to different programs' virtual addresses -- also known as address space -- as different processes are run.