CS-2011, Machine Organization and Assembly Language, D-term 2014

Lab Assignment 4: Option 2
Malloclab

Assigned: April 22, 2014, Due: May 4, 2014
Professor Hugh C. Lauer

Introduction

In this lab you will be writing a dynamic storage allocator for C programs — i.e., your own version of the **malloc**, **free** and **realloc** routines. You are encouraged to explore the design space creatively and implement an allocator that is correct, efficient and fast.

You may work in a group of up to two people. Any clarifications and revisions to the assignment will be posted on the course Web page.

By way of background, study the **malloc** implementation in §8.7 of Kernighan and Ritchie. Also, study the more advanced techniques for implementing **malloc** in §9.9 of Bryant & O'Hallaron. There are no lectures on this subject during this term, so you will have to learn from the reading material and other resources that you can find.

Downloading the assignment

Download the lab from **myWPI** by selecting this link — **malloclab-handout.tar**[1] — and save it to a protected Linux directory in which you plan to work. This will require you to log in with your WPI userID and password. You will also need some trace files to test your **malloc** implementation. These can be downloaded from **Malloclab-traces.zip**.

Alternatively, you may go the course web site, select *Projects* and then select the links for *Malloclab-handout* and *Malloclab-traces* in the *Handouts* column of the table.

Once you have downloaded both files, execute the following command in a shell in your Linux virtual machine to extract the **malloclab** files:–

```
tar xvf malloclab-handout.tar
```

This will cause a number of files to be unpacked into a directory named **malloclab-handout**.

The only file you will be modifying and handing in is **mm.c**. The **mdriver.c** program is a driver program that allows you to evaluate the performance of your solution. Use the command **make** to generate the driver code, and run the driver with the command

[1] It seems that authentication of links into **myWPI** does not work if the link is in a Word document, but it does work if the *exact same* link is in a PDF document or an HTML document. Click on it in the PDF version.

```
./mdriver -V
```

(The **-V** flag displays helpful summary information.)

Looking at the file **mm.c** you'll notice a C structure **team** into which you should insert the requested identifying information about the one or two individuals comprising your programming team. *Do this right away so you don't forget.* When you have completed the lab, you will hand in only one file (**mm.c**), which contains your solution.

How to Work on the Lab

Your dynamic storage allocator will consist of the following four functions, which are declared in **mm.h** and defined in **mm.c**.

```
int   mm_init(void);
void *mm_malloc(size_t size);
void  mm_free(void *ptr);
void *mm_realloc(void *ptr, size_t size);
```

The **mm.c** file in the download file implements the simplest but still functionally correct **malloc** package that we could think of. Using this as a starting place, modify these functions (and possibly define other private **static** functions), so that they obey the following semantics:

- **mm_init:** Before calling **mm_malloc**, **mm_realloc**, or **mm_free**, the application program (i.e., the trace-driven driver program that you will use to evaluate your implementation) calls **mm_init** to perform any necessary initializations, such as allocating the initial heap area. The return value should be **-1** if there was a problem in performing the initialization, **0** otherwise.

- **mm_malloc:** The **mm_malloc** routine returns a pointer to an allocated block payload of at least **size** bytes. The entire allocated block should lie within the heap region and should not overlap with any other allocated chunk.

  We will compare your implementation to the version of **malloc** supplied in the standard *C* library (**libc**). Since the **libc** implementation of **malloc** always returns payload pointers that are aligned to 8 bytes, your **malloc** implementation should also return 8-byte aligned pointers.

- **mm_free:** The **mm_free** routine frees the block pointed to by **ptr**. It returns nothing. This routine is only guaranteed to work when the passed pointer (**ptr**) was returned by an earlier call to **mm_malloc** or **mm_realloc** and has not yet been freed.

- **mm_realloc:** The **mm_realloc** routine returns a pointer to an allocated region of at least **size** bytes with the following constraints.

  - if **ptr** is **NULL**, the call is equivalent to **mm_malloc(size);**
  - if **size** is equal to zero, the call is equivalent to **mm_free(ptr);**
  - if **ptr** is not **NULL**, it must have been returned by an earlier call to **mm_malloc** or **mm_realloc**. The call to **mm_realloc** changes the size of the memory block pointed to by **ptr** (the *old block*) to **size** bytes and returns the address of the new block. Notice that the address of the new block might be the same as the old block, or it might be different, depending on your implementation, the amount of internal fragmentation in the old block, and the size of the **realloc** request.

The contents of the new block are the same as those of the old **`ptr`** block, up to the minimum of the old and new sizes. Everything else is uninitialized. For example, if the old block is 8 bytes and the new block is 12 bytes, then the first 8 bytes of the new block are identical to the first 8 bytes of the old block and the last 4 bytes are uninitialized. Similarly, if the old block is 8 bytes and the new block is 4 bytes, then the contents of the new block are identical to the first 4 bytes of the old block.

These semantics match the the semantics of the corresponding **`libc`** versions of **`malloc`**, **`realloc`**, and **`free`** routines. Type **`man malloc`** to the shell for complete documentation.

Heap Consistency Checker

Dynamic memory allocators are notoriously tricky beasts to program correctly and efficiently. They are difficult to program correctly because they involve a lot of untyped pointer manipulation. You will find it very helpful to write a heap checker that scans the heap and checks it for consistency.

Some examples of what a heap checker might check are:–

- Is every block in the free list marked as free?
- Are there any contiguous free blocks that somehow escaped coalescing?
- Is every free block actually in the free list?
- Do the pointers in the free list point to valid free blocks?
- Do any allocated blocks overlap?
- Do the pointers in a heap block point to valid heap addresses?

Your heap checker will consist of the function **`int mm_check(void)`** in **`mm.c`**. It should check any invariants or consistency conditions you consider prudent. It returns a nonzero value if and only if your heap is consistent. You are not limited to the listed suggestions nor are you required to check all of them. You are encouraged to print out error messages when **`mm_check`** fails.

This consistency checker is for your own debugging during development. When you submit **`mm.c`**, make sure to remove any calls to **`mm_check`** as they will slow down your throughput. Style points will be given for your **`mm_check`** function. Make sure to put in comments and document what you are checking.

Support Routines

The **`memlib.c`** package simulates the memory system for your dynamic memory allocator. You can invoke the following functions in **`memlib.c`**:

- **`void *mem_sbrk(int incr)`**: Expands the heap by **`incr`** bytes, where **`incr`** is a positive non-zero integer. It returns a generic pointer to the first byte of the newly allocated heap area. The semantics are identical to the Unix **`sbrk`** function, except that **`mem_sbrk`** accepts only a positive non-zero integer argument.
- **`void *mem_heap_lo(void)`**: Returns a generic pointer to the first byte in the heap.
- **`void *mem_heap_hi(void)`**: Returns a generic pointer to the last byte in the heap.

- **`size_t mem_heapsize(void)`**: Returns the current size of the heap in bytes.
- **`size_t mem_pagesize(void)`**: Returns the system's page size in bytes (4K on Linux systems).

The Trace-driven Driver Program

The driver program **`mdriver.c`** in the **`malloclab-handout.tar`** distribution tests your **`mm.c`** package for correctness, space utilization, and throughput. The driver program is controlled by a set of *trace files* that are included in **`Malloclab-traces.zip`**. Each trace file contains allocate, reallocate, and free commands that cause the driver to call your **`mm_malloc`**, **`mm_realloc`**, and **`mm_free`** routines in some sequence. The driver and the trace files are the same ones we will use when we grade your **`mm.c`** submission.

Unzip **`Malloclab-traces.zip`** to a suitable directory so that it can be accessed by the driver. The driver **`mdriver.c`** accepts the following command line arguments:–

- **`-t <tracedir>`**: Look for the default trace files in directory **`tracedir`** instead of the default directory defined in **`config.h`**.
- **`-f <tracefile>:`** Use one particular **`tracefile`** for testing instead of the default set of tracefiles.
- **`-h`**: Print a summary of the command line arguments.
- **`-l`**: Run and measure **`libc`** malloc in addition to the student's malloc package.
- **`-v`**: Verbose output. Print a performance breakdown for each tracefile in a compact table.
- **`-V`**: More verbose output. Prints additional diagnostic information as each trace file is processed. Useful during debugging for determining which trace file is causing your malloc package to fail.

Programming Rules

- You should not change any of the interfaces in **`mm.c`**.
- You should not invoke any memory-management related library calls or system calls. In particular, you may not call the system versions of **`malloc`**, **`calloc`**, **`free`**, **`realloc`**, **`sbrk`**, **`brk`** or any variants of these calls in your code.
- You are not allowed to define any global or **`static`** compound data structures such as arrays, **`struct`**s, trees, or **`lists`** in your **`mm.c`** program. However, you *are* allowed to declare global scalar variables such as integers, floats, and pointers in **`mm.c`**. [2]
- For consistency with the **`libc malloc`** package, which returns blocks aligned on 8-byte boundaries, your allocator must always return pointers that are aligned to 8-byte boundaries. The driver will enforce this requirement for you.

[2] Note that the implementation of **`malloc()`** published in Kernighan and Ritchie *does* use a global **`struct`**!

Evaluation

You will receive **zero points** if you break any of the rules or your code is buggy and crashes the driver. Otherwise, your grade will be calculated as follows:

- *Correctness (20 points).* You will receive full points if your solution passes the correctness tests performed by the driver program. You will receive partial credit for each correct trace.
- *Performance (35 points).* Two performance metrics will be used to evaluate your solution:
  - ◦ *Space utilization*: The peak ratio between the aggregate amount of memory used by the driver (i.e., allocated via **`mm_malloc`** or **`mm_realloc`** but not yet freed via **`mm_free`**) and the size of the heap used by your allocator. The optimal ratio equals to 1. You should find good policies to minimize fragmentation in order to make this ratio as close as possible to the optimal.
  - ◦ *Throughput*: The average number of operations completed per second.

  The driver program summarizes the performance of your allocator by computing a performance index, P, which is a weighted sum of the space utilization and throughput

  $$P = wU + (1-w)\min\left(1, \frac{T}{T_{libc}}\right)$$

  where U is your space utilization, T is your throughput, and T_{libc} is the estimated throughput of **`libc`** malloc on your system on the default traces.[3] The performance index favors space utilization over throughput, with a default of w=0.6.

  Observing that both memory and CPU cycles are expensive system resources, we adopt this formula to encourage balanced optimization of both memory utilization and throughput. Ideally, the performance index will reach $P=w+(1-w)=1$ or 100%. Since each metric will contribute at most w and $1-w$ to the performance index, respectively, you should not go to extremes to optimize either the memory utilization or the throughput only. To receive a good score, you must achieve a balance between utilization and throughput.
- Style (10 points).
  - ◦ Your code should be decomposed into functions and use as few global variables as possible.
  - ◦ Your code should begin with a header comment that describes the structure of your free and allocated blocks, the organization of the free list, and how your allocator manipulates the free list. each function should be preceeded by a header comment that describes what the function does.
  - ◦ Each subroutine should have a header comment that describes what it does and how it does it.
  - ◦ Your heap consistency checker **`mm_check`** should be thorough and well-documented.

  You will be awarded 5 points for a good heap consistency checker and 5 points for good program structure and comments.

[3] The value for T_{libc} is a constant in the driver (1000 Kops/s) that your instructor established when configuring the program. This gives about half the possible throughput points for the K&R **`malloc`** algorithm.

- **README** file (10 points): You must also submit a **README** file in **.doc**, **.docx**, **pdf**, or.**txt** format that describes your data structures and the algorithms of your implementation. In particular, you must describe the design choices you made among the competing demands of a good **malloc** package. For example, does your package coalesce adjacent free nodes? If so, how does it discover that the adjacent nodes are free? If not, what benefit is gained by not doing so?

Submission Instructions

When you are ready to submit, rename your **mm.c** file to **userID-mm.c**, where userID is your WPI login ID. Then submit this file and your **README** file to the *Malloclab* project in the web-based *Turnin* system at

http://turnin.cs.wpi.edu

Hints

- *Use the* **mdriver -f** *option.* During initial development, using tiny trace files will simplify debugging and testing. We have included two such trace files (**short1,2-bal.rep**) that you can use for initial debugging.
- *Use the* **mdriver -v** *and* **-V** *options.* The **-v** option will give you a detailed summary for each trace file. The **-V** will also indicate when each trace file is read, which will help you isolate errors.
- *Compile with* **gcc -g** *and use a debugger.* A debugger will help you isolate and identify out of bounds memory references.
- *Understand every line of the malloc implementation in the textbook*. The textbook has a detailed example of a simple allocator based on an implicit free list. Use this is a point of departure. Don't start working on your allocator until you understand everything about the simple implicit list allocator.
- *Encapsulate your pointer arithmetic in C preprocessor macros.* Pointer arithmetic in memory managers is confusing and error-prone because of all the casting that is necessary. You can reduce the complexity significantly by writing macros for your pointer operations. See the text for examples.
- *Do your implementation in stages.* The first 9 traces contain requests to **malloc** and **free**. The last 2 traces contain requests for **realloc**, **malloc**, and **free**. We recommend that you start by getting your **malloc** and **free** routines working correctly and efficiently on the first 9 traces. Only then should you turn your attention to the **realloc** implementation. For starters, build **realloc** on top of your existing **malloc** and **free** implementations. But to get really good performance, you will need to build a stand-alone **realloc**.
- *Use a profiler.* You may find the **gprof** tool helpful for optimizing performance.
- *Start early!* It is possible to write an efficient **malloc** package with a few pages of code. However, we can guarantee that it will be some of the most difficult and sophisticated code you have written so far in your career. So start early, and good luck!

Linking and Loading

Professor Hugh C. Lauer
CS-2011, Machine Organization and Assembly Language

(Slides include copyright materials from *Computer Systems: A Programmer's Perspective*, by Bryant and O'Hallaron, and from *The C Programming Language*, by Kernighan and Ritchie)

Today

- Linking
- Case study: Library interpositioning

Reading Assignment: §7.1 – §7.12

Example C Program

main.c

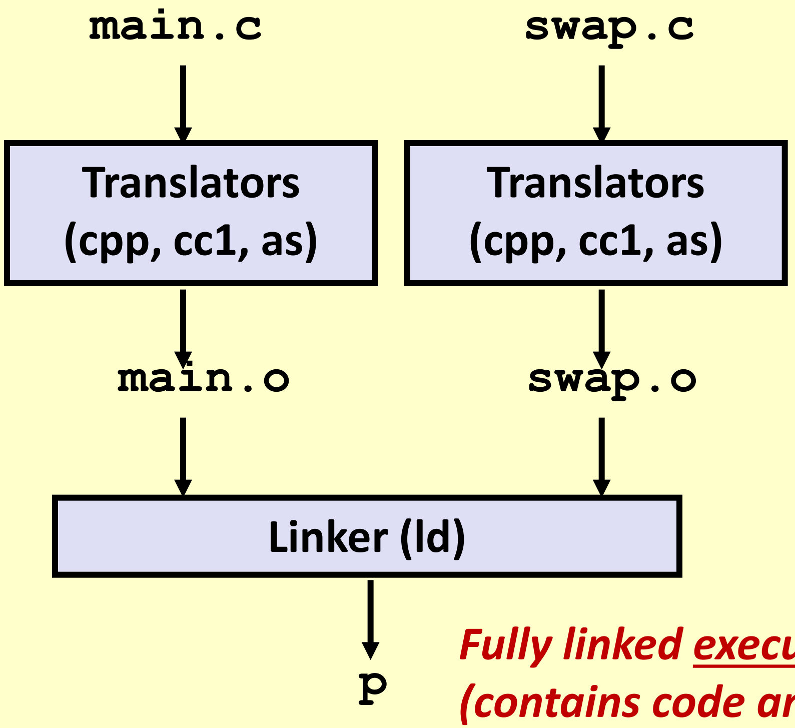
```
int buf[2] = {1, 2};

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*:
  - `linux> gcc -O2 -g -o p main.c swap.c`
  - `linux> ./p`



Source files

Separately compiled relocatable object files

Fully linked executable object file (contains code and data for all functions defined in `main.c` and `swap.c`)

Why Linkers?

- **Reason 1: Modularity**
  - Program can be written as a collection of smaller source files, rather than one monolithic mass.
  - Especially amenable to team development
  - Can build libraries of common functions (more on this later)
    - e.g., Math library, standard C library

Why Linkers? (continued)

- **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?

- **Symbol Resolution**
  - I.e., connect declared/defined objects with references to them in other modules

- **Relocation**
  - I.e., reposition code within executable image and change values of internal pointers to match

What Do Linkers Do?

- **Step 1. Symbol resolution**
  - Programs define and reference *symbols* (variables and functions):
    - `void swap() {…}    /* define symbol "swap" */`
    - `swap();            /* reference symbol "swap" */`
    - `int *xp = &x;      /* define symbol "xp", reference symbol "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 whatever the symbol refers to.
  - Linker associates each symbol reference with exactly one symbol definition.

What Do Linkers Do? (continued)

- **Step 2. Relocation**
  - Merge separate code and data sections into single sections
  - Relocate symbols from their relative locations in the `.o` files to their final absolute memory locations in the executable.
  - Update 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)

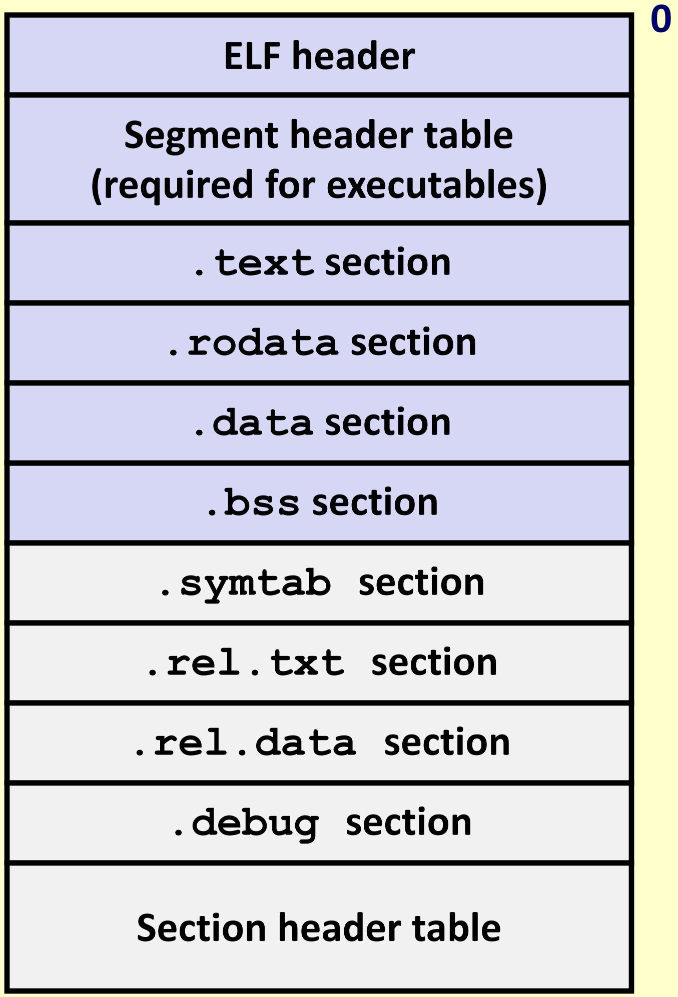
- **Standard binary format for object files**
  - Originally proposed by AT&T System V Unix
  - Later adopted by BSD Unix variants and Linux

- **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, vtables, etc., ...
- **`.data` section**
  - Initialized global & static variables
- **`.bss` section**
  - Uninitialized global & static variables
  - "Block Storage Start"
  - "Better Save Space"
  - Has section header but occupies no space



0

ELF header

Segment header table
(required for executables)

`.text` section

`.rodata` section

`.data` section

`.bss` section

`.symtab` section

`.rel.txt` section

`.rel.data` section

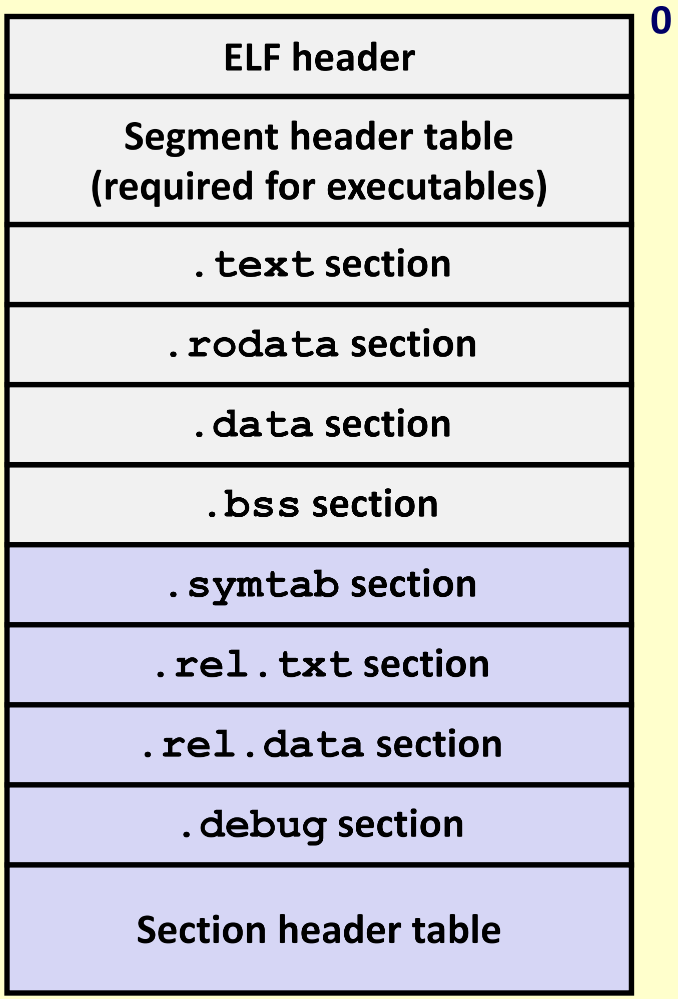
`.debug` section

Section header table

See §7.4, p. 659

ELF Object File Format (continued)

- **`.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.
- **`.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



0

ELF header

Segment header table
(required for executables)

`.text` section

`.rodata` section

`.data` section

`.bss` section

`.symtab` section

`.rel.txt` section

`.rel.data` section

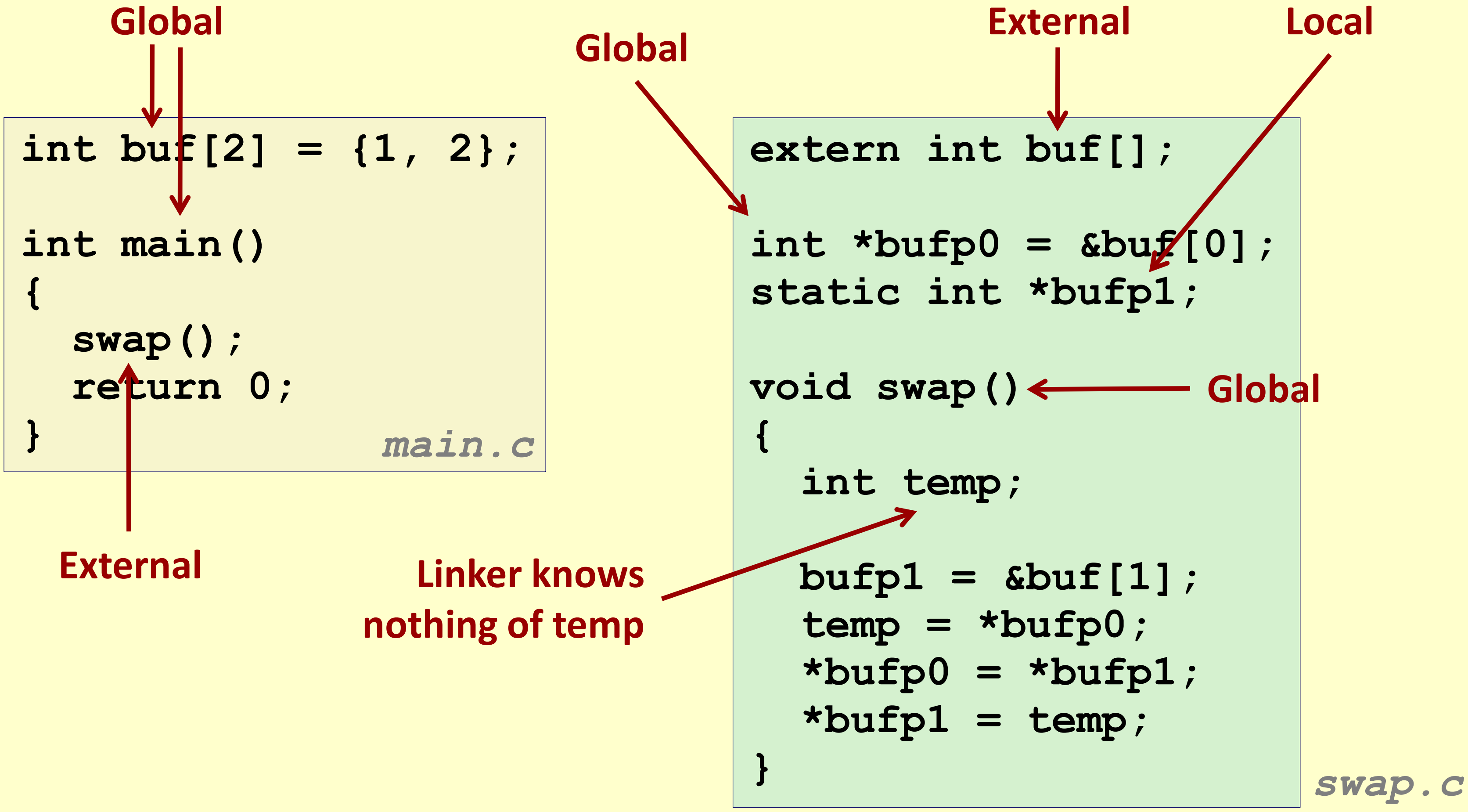
`.debug` section

Section header table

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

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

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

main.c

External

Global

External

Local

```
extern int buf[];

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

void swap()
{
   int temp;

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

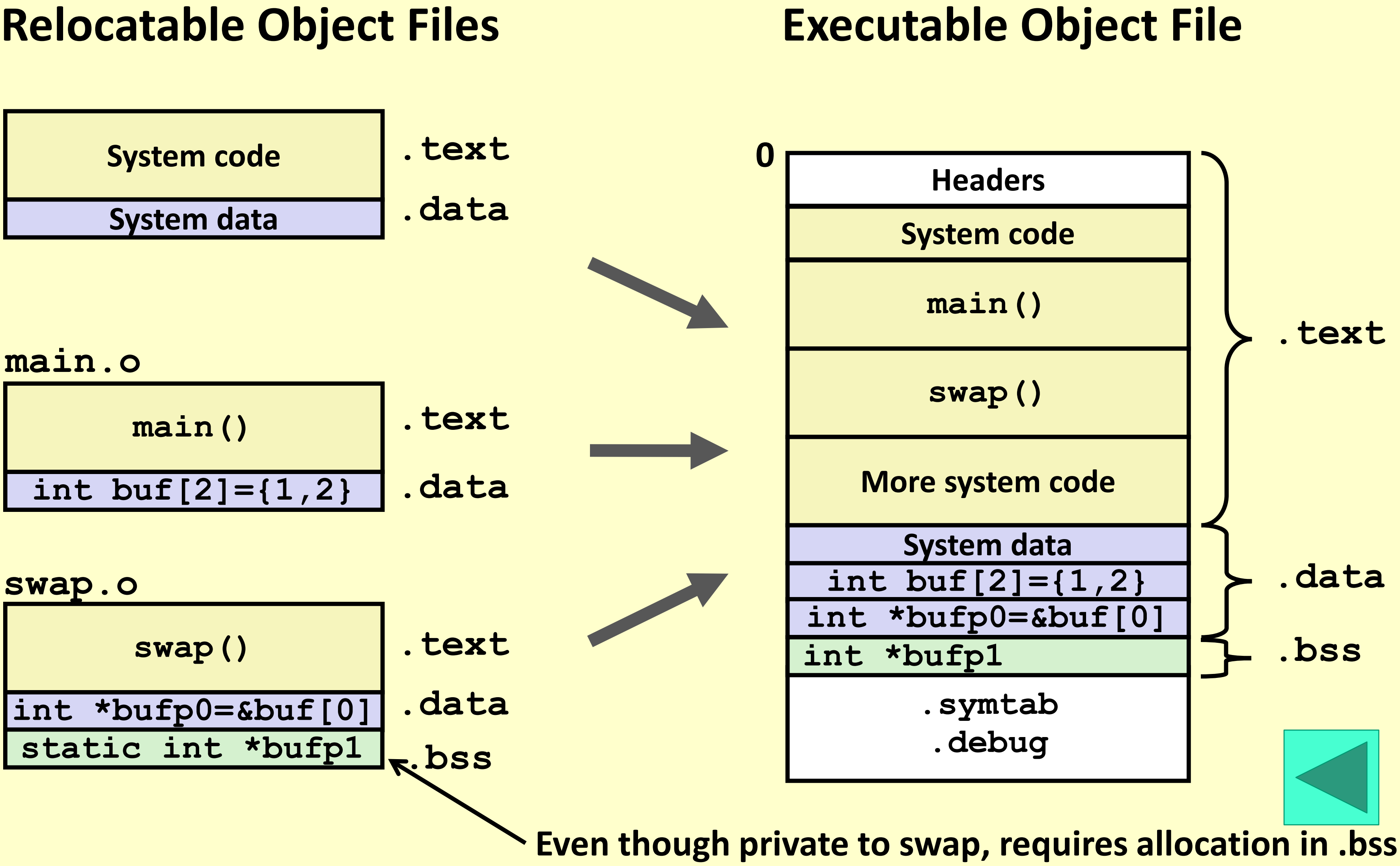
Global

Linker knows nothing of temp

swap.c

Questions?

Relocating Code and Data



Even though private to swap, requires allocation in .bss

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
```

Source: `objdump -r -d`

```
Disassembly of section .data:

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

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 b0 48 83 08       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
```

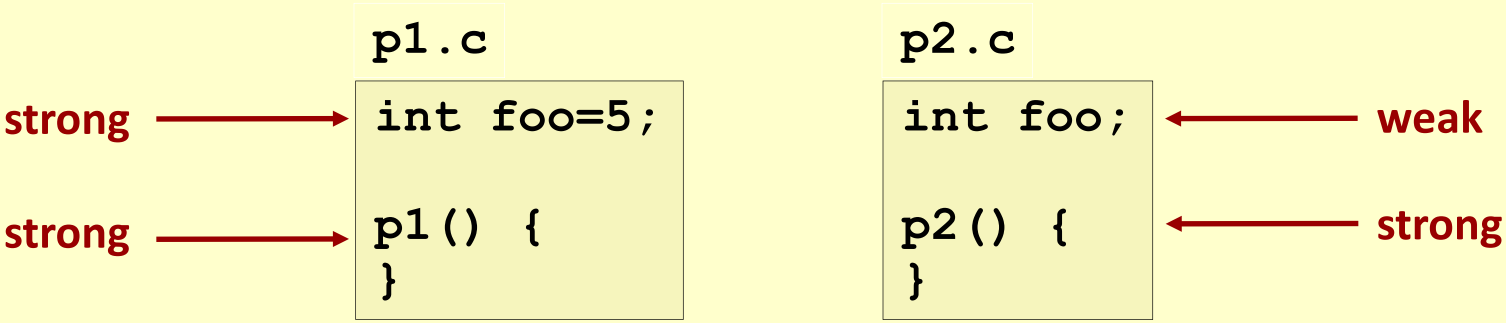
Executable After Relocation (.data)

```
Disassembly of section .data:

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

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

Questions?

Strong and Weak Symbols

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



p1.c

```
int foo=5;

p1() {
}
```

strong

strong

p2.c

```
int foo;

p2() {
}
```

weak

strong

Linker's Symbol Rules

- **Rule 1: Multiple strong symbols *of same name* are not allowed**
  - Each item can be defined only once
  - Otherwise: Linker error
- **Rule 2: Given one strong symbol and multiple weak symbols of same name, 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

```
#include "global.h"

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

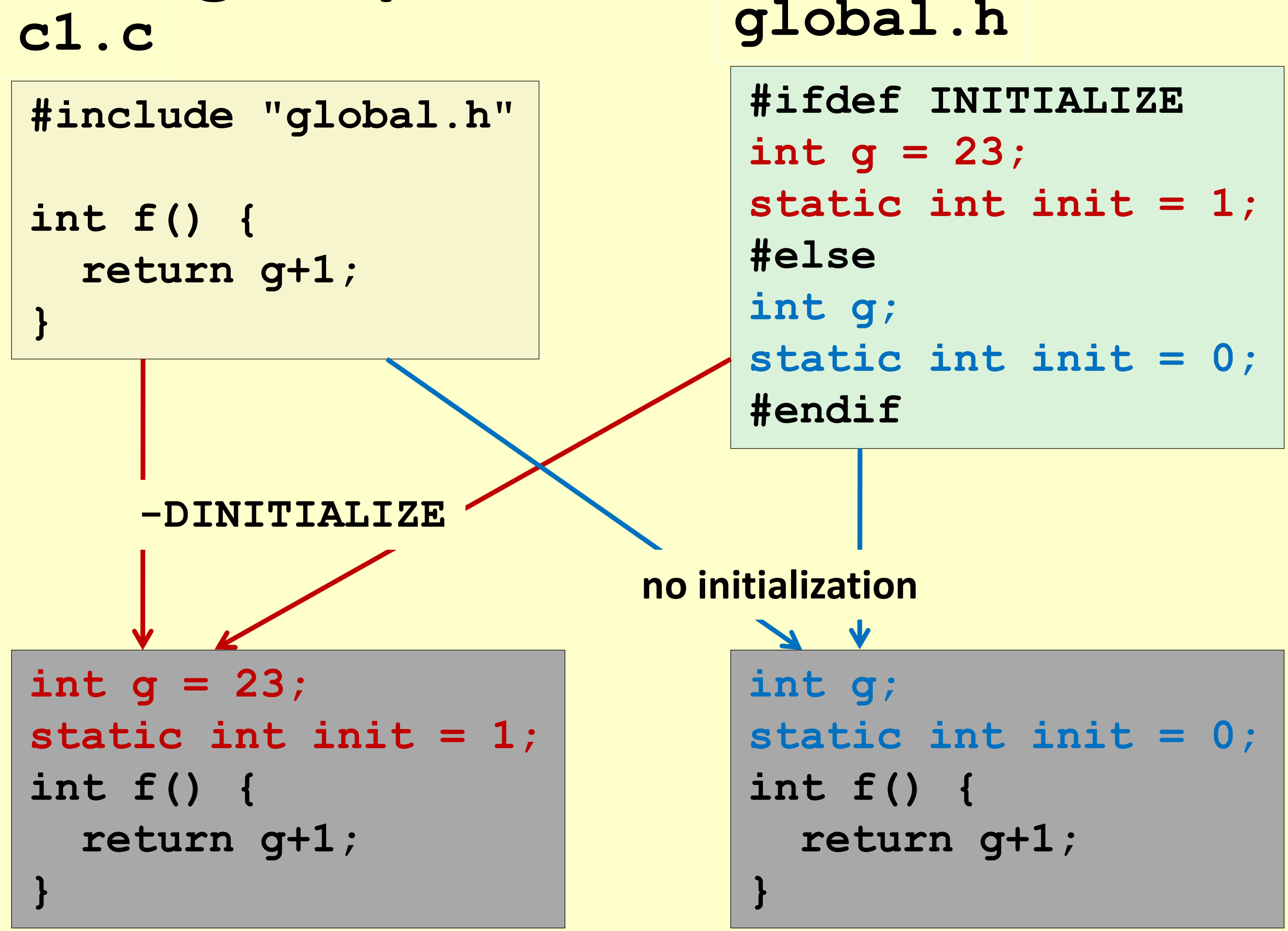
c2.c

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

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

global.h

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

Running Preprocessor



c1.c

```
#include "global.h"

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

global.h

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

-DINITIALIZE

no initialization

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

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

#include causes C preprocessor to insert file verbatim

Role of .h Files (continued)

c1.c

```
#include "global.h"

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

global.h

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

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` *whenever* you want access to an external global variable

Questions?

Packaging Commonly Used Functions

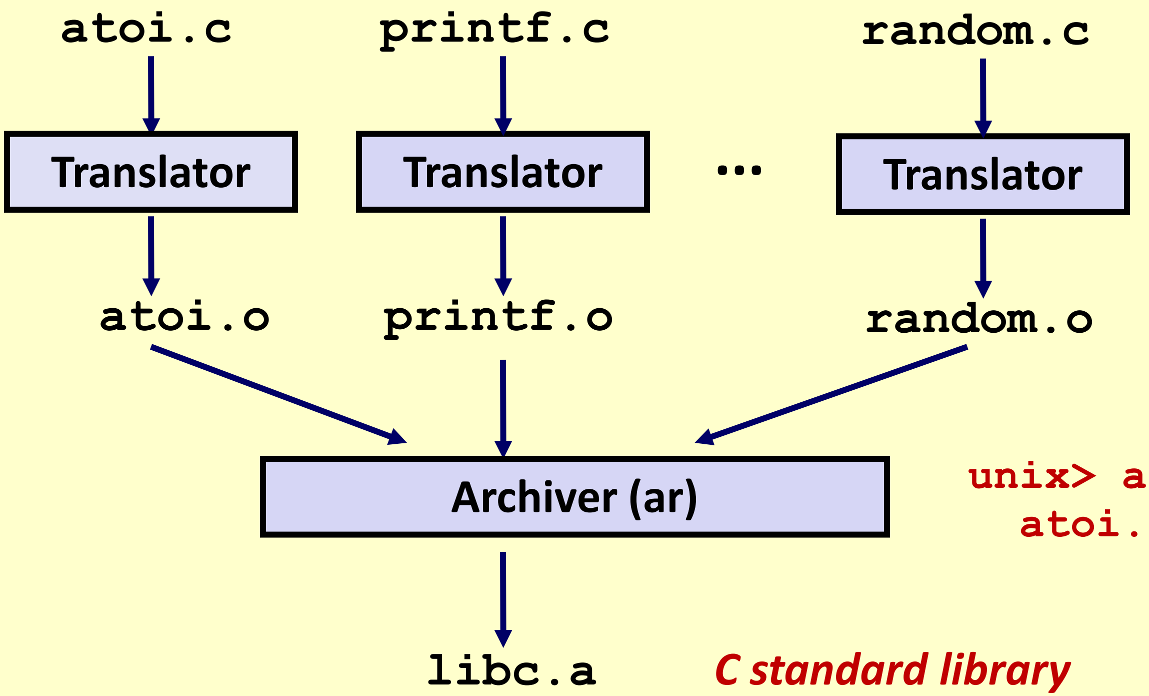
- **How to package functions commonly used by programmers?**
  - Math, I/O, memory management, string manipulation, etc.

- **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
```

C standard library

- 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

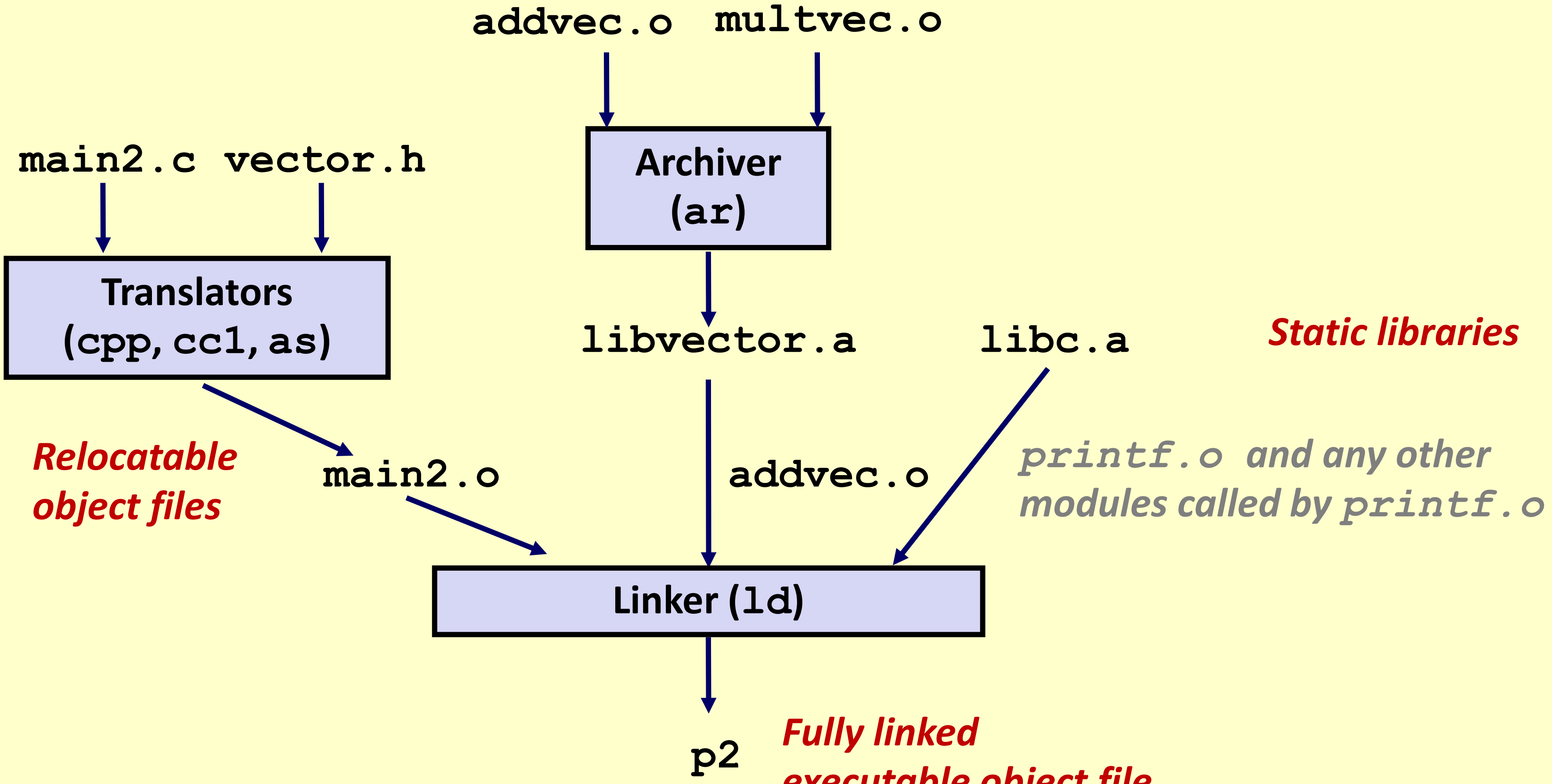
`libm.a` (the C math library)

- 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

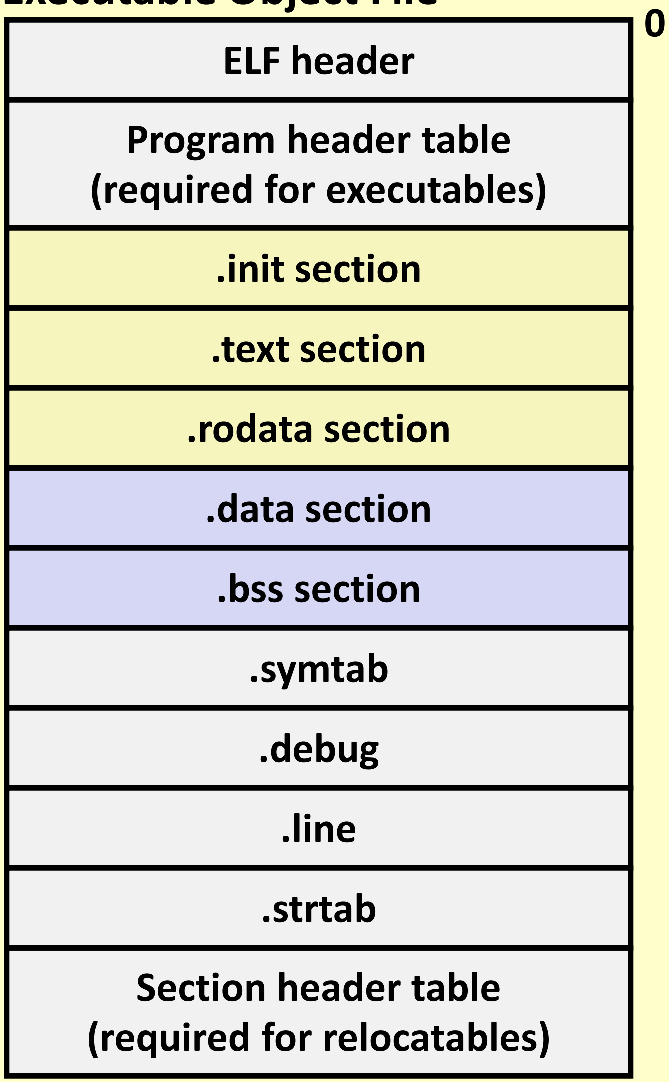
- **Linker's algorithm for resolving external references:**
  - Scan `.o` files and `.a` files in the command line order.
  - During the scan, keep a list of the current unresolved references.
  - As each new `.o` or `.a` file, *obj*, is encountered, try to resolve each unresolved reference in the list against the symbols defined in *obj*.
  - If any entries in the unresolved list at end of scan, then error.

- **Problem:**
  - Command line order matters!
  - Moral: put libraries at the end of the command line.

```
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

Executable Object File



0

ELF header

Program header table
(required for executables)

.init section

.text section

.rodata section

.data section

.bss section

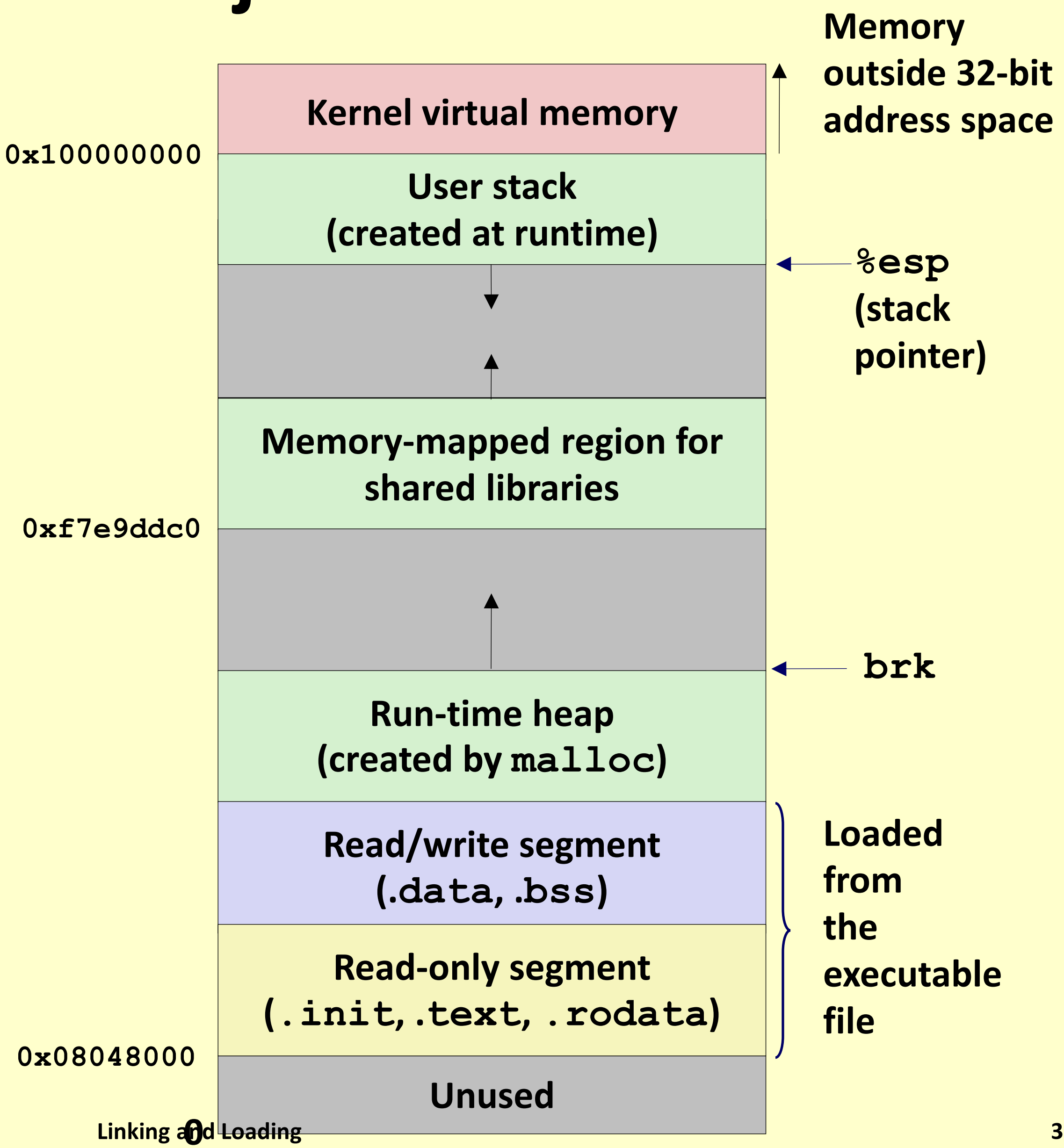
.symtab

.debug

.line

.strtab

Section header table
(required for relocatables)



Memory outside 32-bit address space

0x100000000

Kernel virtual memory

User stack
(created at runtime)

%esp
(stack pointer)

Memory-mapped region for shared libraries

0xf7e9ddc0

brk

Run-time heap
(created by malloc)

Read/write segment
(.data, .bss)

Read-only segment
(.init, .text, .rodata)

Loaded from the executable file

0x08048000

Unused

0

Shared Libraries

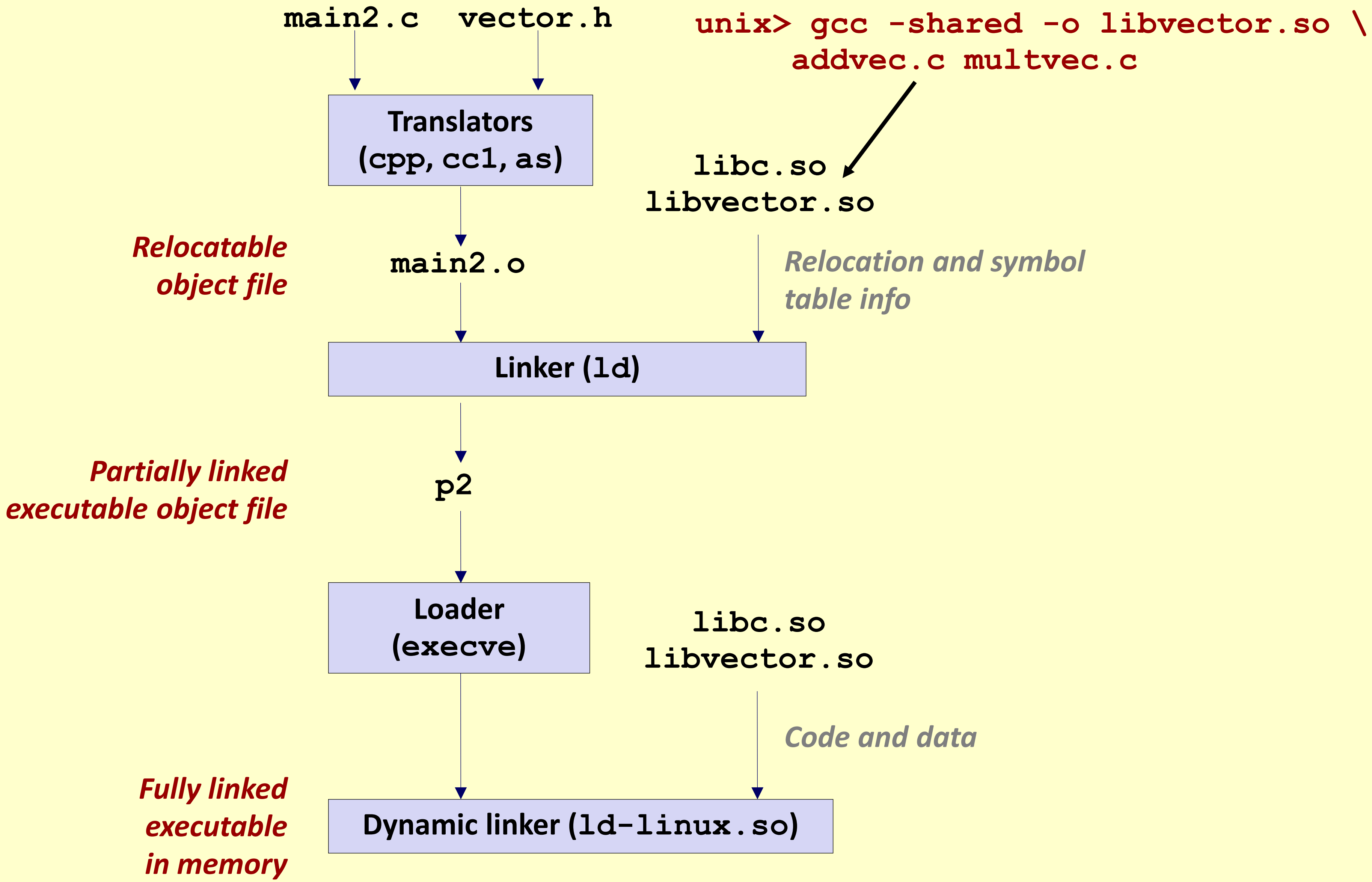
- **Static libraries have the following disadvantages:**
  - Duplication in the stored executables (every function need std **libc**)
  - Duplication in the running executables
  - Minor bug fixes of system libraries require each application to explicitly relink

- **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 (continued)

- **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.
    - Runtime library interpositioning.
- **Shared library routines can be shared by multiple processes.**
  - More on this when we learn about virtual memory (in OS course!)

Dynamic Linking at Load-time



```
unix> gcc -shared -o libvector.so \
    addvec.c multvec.c
```

main2.c
vector.h
Translators (cpp, cc1, as)
libc.so
libvector.so
Relocatable object file
main2.o
Relocation and symbol table info
Linker (ld)
Partially linked executable object file
p2
Loader (execve)
libc.so
libvector.so
Code and data
Fully linked executable in memory
Dynamic linker (ld-linux.so)

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;
}
```

Today

- Linking
- Case study: Library interpositioning

Case Study: Library Interpositioning

- **Library interpositioning : powerful linking technique that allows programmers to intercept calls to arbitrary functions**

- **Interpositioning can occur at:**
  - Compile time: When the source code is compiled
  - Link time: When the relocatable object files are statically linked to form an executable object file
  - Load/run time: When an executable object file is loaded into memory, dynamically linked, and then executed.

Some Interpositioning Applications

- **Security**
  - Confinement (sandboxing)
    - Interpose calls to libc functions.
  - Behind the scenes encryption
    - Automatically encrypt otherwise unencrypted network connections.
- **Monitoring and Profiling**
  - Count number of calls to functions
  - Characterize call sites and arguments to functions
  - Malloc tracing
    - Detecting memory leaks
    - **Generating address traces**

Example program

```c
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>

int main()
{
    free(malloc(10));
    printf("hello, world\n");
    exit(0);
}
                                  hello.c
```

- **Goal: trace the addresses and sizes of the allocated and freed blocks, without modifying the source code.**
- **Three solutions: interpose on the `lib` `malloc` and `free` functions at**
  - compile time,
  - link time, and/or
  - load/run time.

Compile-time Interpositioning

```
#ifdef COMPILETIME
/* Compile-time interposition of malloc and free using C
 * preprocessor. A local malloc.h file defines malloc (free)
 * as wrappers mymalloc (myfree) respectively.
 */

#include <stdio.h>
#include <malloc.h>

/*
 * mymalloc - malloc wrapper function
 */
void *mymalloc(size_t size, char *file, int line)
{
    void *ptr = malloc(size);
    printf("%s:%d: malloc(%d)=%p\n", file, line, (int)size,
ptr);
    return ptr;
}
```

mymalloc.c

Compile-time Interpositioning

```
#define malloc(size) mymalloc(size, __FILE__, __LINE__ )
#define free(ptr) myfree(ptr, __FILE__, __LINE__ )

void *mymalloc(size_t size, char *file, int line);
void myfree(void *ptr, char *file, int line);
```

malloc.h

```
linux> make helloc
gcc -O2 -Wall -DCOMPILETIME -c mymalloc.c
gcc -O2 -Wall -I. -o helloc hello.c mymalloc.o
linux> make runc
./helloc
hello.c:7: malloc(10)=0x501010
hello.c:7: free(0x501010)
hello, world
```

Link-time Interpositioning

```
#ifdef LINKTIME
/* Link-time interposition of malloc and free using the
static linker's (ld) "--wrap symbol" flag. */

#include <stdio.h>

void *__real_malloc(size_t size);
void __real_free(void *ptr);

/*
 * __wrap_malloc - malloc wrapper function
 */
void *__wrap_malloc(size_t size)
{
    void *ptr = __real_malloc(size);
    printf("malloc(%d) = %p\n", (int)size, ptr);
    return ptr;
}
```

mymalloc.c

Link-time Interpositioning

```
linux> make hellol
gcc -O2 -Wall -DLINKTIME -c mymalloc.c
gcc -O2 -Wall -Wl,--wrap,malloc -Wl,--wrap,free \
-o hellol hello.c mymalloc.o
linux> make runl
./hellol
malloc(10) = 0x501010
free(0x501010)
hello, world
```

- **The “`-Wl`” flag passes argument to linker**
- **Telling linker “`--wrap,malloc`” tells it to resolve references in a special way:**
  - Refs to **`malloc`** should be resolved as **`__wrap_malloc`**
  - Refs to **`__real_malloc`** should be resolved as **`malloc`**

Load/Run-time Interpositioning

```
#ifdef RUNTIME
 /* Run-time interposition of malloc and free based on
 * dynamic linker's (ld-linux.so) LD_PRELOAD mechanism */
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

void *malloc(size_t size)
{
    static void *(*mallocp)(size_t size);
    char *error;
    void *ptr;

    /* get address of libc malloc */
    if (!mallocp) {
        mallocp = dlsym(RTLD_NEXT, "malloc");
        if ((error = dlerror()) != NULL) {
            fputs(error, stderr);
            exit(1);
        }
    }
    ptr = mallocp(size);
    printf("malloc(%d) = %p\n", (int)size, ptr);
    return ptr;
}
```

mymalloc.c

Load/Run-time Interpositioning

```
linux> make hellor
gcc -O2 -Wall -DRUNTIME -shared -fPIC -o mymalloc.so mymalloc.c
gcc -O2 -Wall -o hellor hello.c
linux> make runr
(LD_PRELOAD="/usr/lib64/libdl.so ./mymalloc.so" ./hellor)
malloc(10) = 0x501010
free(0x501010)
hello, world
```

- **The `LD_PRELOAD` environment variable tells the dynamic linker to resolve unresolved refs (e.g., to `malloc`) by looking in `libdl.so` and `mymalloc.so` first.**
  - **`libdl.so`** necessary to resolve references to the **`dlopen`** functions.

Interpositioning Recap

- **Compile Time**
  - Apparent calls to malloc/free get macro-expanded into calls to mymalloc/myfree
- **Link Time**
  - Use linker trick to have special name resolutions
    - malloc → __wrap_malloc
    - __real_malloc → malloc
- **Compile Time**
  - Implement custom version of malloc/free that use dynamic linking to load library malloc/free under different names

Questions?