CS 33: Introduction to Computer Organization

Week 10, Part 1

Agenda

- Info regarding final
- A couple of questions from the previous midterm.
- Virtual Memory
- Linking
- Exceptional Control Flow

Final

- Admin stuff:
  - Begins at 8:00 AM (sorry)
  - Expected to take the full 3 hours (sorry)
  - BH 3400 (sorry)
  - Please arrive about 5 minutes early. (sorry)

Final

- Mostly focused on the last third, but is cumulative:
  - I/O
  - Virtual Memory
  - Linking
  - Exceptional Control Flow
- ~40% material after the second midterm
- ~60% previous material
- 100% despair

Previous Midterm Q4

- Suppose we change the Core i7’s virtual addresses to use "huge" pages, i.e., pages of size 1 GiB. We alter the rest of the implementation as little as possible: for example, we don’t change the page table size.
- 4a (10 minutes). What should virtual addresses look like with under this regime? In other words, what components would virtual addresses be broken up into, compared to the components normally used in the Core i7?

Previous Midterm Q4

- 4b (10 minutes). Give two performance advantages that come from having huge pages, as opposed to the usual 4 KiB pages. One should be a time advantage, the other a space advantage. Briefly explain.
- 4c (10 minutes). Give two performance disadvantages, again one in time and one in space. Briefly explain.

Previous Midterm Q4

- Addresses were originally 48-bits long.
- Page Size is usually 4 KiB or 2^12 bytes.
- Now, pages are 1 GiB or 2^30 bytes.
- How many bits are required for the virtual page offset?
  –
- How many bits are required for the virtual page number?
  –

Previous Midterm Q4

- Addresses were originally 48-bits long.
- Page Size is usually 4 KiB or 2^12 bytes.
- Now, pages are 1 GiB or 2^30 bytes.
- How many bits are required for the virtual page offset?
  - 30
- How many bits are required for the virtual page number?
  - 18

Previous Midterm Q4

- Time Advantage?
  - Applications that access large sequential data structures will benefit from not needing to constantly swap in new pages for that data structure.
  - Say a single array spans 1 GiB. To iterate through this using the normal Nehalem 4 KiB page size (2^12 bytes per page), you'd need to page fault 2^18 times to pull it all into memory.
  - With the 1GiB huge page, it would take only page fault.
  - Disk accesses are incredibly slow so this could be a win.

Previous Midterm Q4

- Space Advantage?
  - Smaller page tables.
  - We will now only need 2^18 entries. Previously since the VPO was 12 bits, we needed 2^36 entries. Each entry governs more data.
  - Additionally, consider 32-bit physical addresses. In the 2^12 byte page size case, the physical page number would require 20-bits. However, if the VPO requires 30-bits, then so does the PPO. Thus, the physical page number in the new case would be 2 bits. Each entry is also smaller.

Previous Midterm Q4

- Time Disadvantage?
  - Because each page is huge, actually copying a single page from disk to RAM will take much longer than in the case of a conservatively sized page.

Previous Midterm Q4

- Space Disadvantage?
  - Say we have 4 MiB of data that we’re interested in, spread across 4 different pages (ie, the text segment has 1 MiB we want to use, the data segment has 1 MiB, etc.)
  - To pull this all into RAM, we need to use of 4 GiB of physical memory even though we were only interested in 4 MiB total.

Previous Midterm Q5

5 (10 minutes). If you compile the following GNU/Linux x86 assembly-language program:

```
        .globl main
  main:
        movl $amusing, p
        call *p
  amusing:
        .byte 15
        .byte -57
        .byte -16
        .bss
  p:
        .zero 4
```

...with 'gcc -m32' and run it on the SEASnet GNU/Linux servers, the program will behave this way:

```
 Illegal instruction (core dumped)
```

This program has received signal 4 (SIGILL, Illegal instruction) and has dumped core.

Previous Midterm Q5

- Crash course in compiler produced assembler directives:
- The instructions are read sequentially. The assembler has a “current section” and “current location into memory” that it will add each item to as it does so.
- The compiler doesn't run these instructions. But it does interpret them to produce a relocatable object file.
- Items marked like “blah:” are labels, effectively pointers. Items marked like “.blah” are assembler directives, telling the assembler to

Previous Midterm Q5

1 `.globl main`
  - defines “main” as a global name

2 `main:`
  - The current section is in “text”. The label main will point to a function.

3 `movl $amusing, p`
  - Move the pointer (not the value of) “amusing” into the data at location p.

4 `call *p`
  - Dereference the pointer “p”, and call the value at pointer “p” as a function.

- Note the inconsistent syntax. Say “amusing” is the pointer 0x123456 and “p” is the pointer 0xAAAAAA
  - `movl $amusing, p == movl $0x123456, ($0xAAAAAA)`
  - `movl amusing, p` is not valid because it is the equivalent of `movl ($0x123456), ($0xAAAAAA)`

Previous Midterm Q5

- However:
  - call p == call 0xAAAAAA
  - call *p == call ($0xAAAAAA)
- 5 amusing:
  - Define a new label "amusing" in the text section.

```
6       .byte 15
7       .byte –57
8       .byte –16
```

  - The bytes at location "amusing" are 15, -57, and -16. Since this is the text section, this is treated as instruction bytes

```
9       .bss
```

  - Switch to the .bss section. Anything after this is .bss

```
10 p:
11      .zero 4
```

  - *p or (p) is four bytes of 0x00

Previous Midterm Q5

- Thus, the original code:

```
main:
      movl $amusing, p
      call *p
```

- Places the address of amusing in p. Originally *p == 0x00000000. Now, *p == <ADDRESS OF AMUSING>, say 0x123456.
- Then, we call *p, which means:
  - call 0x123456 <amusing>
- However, the byte instruction at amusing is not valid. They're arbitrary bytes. Hence, SIGILL (illegal instruction).

Previous Midterm Q5

- Suppose we remove line 3 from the program. How will it behave instead? Give the sequence of instructions that it will execute, and the behavior the invoker will observe.

```
  main:
4       call *p
```

- *p == 0x00000000 since it it initialized as ".zero 4"
- Then, we call *p, which means:
  - call 0x00000000
- We attempt to call the null pointer. The null pointer is invalid memory so instead the error is a SIGSEGV or a segmentation fault.

Virtual Memory

- A comment regarding the TLB.
- Recall that the TLB is an additional cache that allows us to store page table entries.
- Unlike the page table, the TLB is a physical component.
- Is there going to be a problem with that?

Virtual Memory

- We will have one TLB shared among all of the processes.
- You index into a TLB using the virtual address.
- Each process shares the virtual address space.
- This means that a TLB entry that is valid for one process may *appear* to be valid for another process if they happen to issue the same virtual address

Virtual Memory

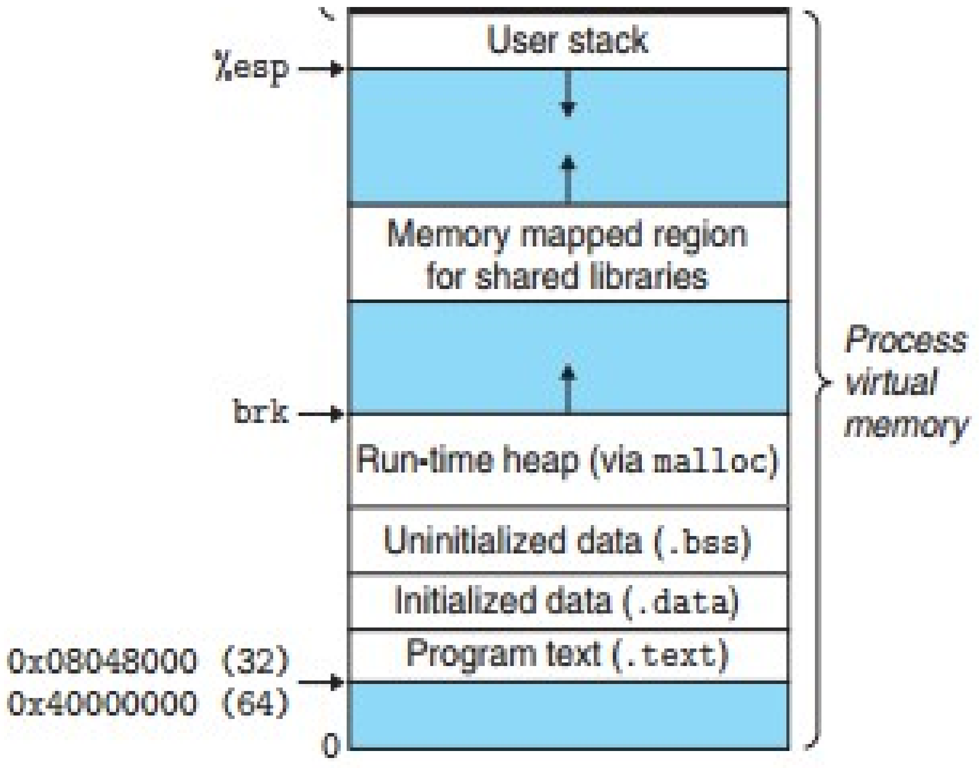
- If left unchecked, this could be very bad. We could get false positive hits in the TLB and just keep going with the wrong addresses.
- The simple solution (as unfortunate as it is) is to flush the TLB after every context switch.
- An alternative would be to store information in the TLB as to which process an entry corresponds to.

Virtual Memory

- The common addressable space consists of “virtual addresses”.
- This means these addresses do not directly correspond to addresses of physical memory.
- The virtual address space of a process can be split up and divided into arbitrary areas and pages; virtual to physical mapping will take care of the rest.

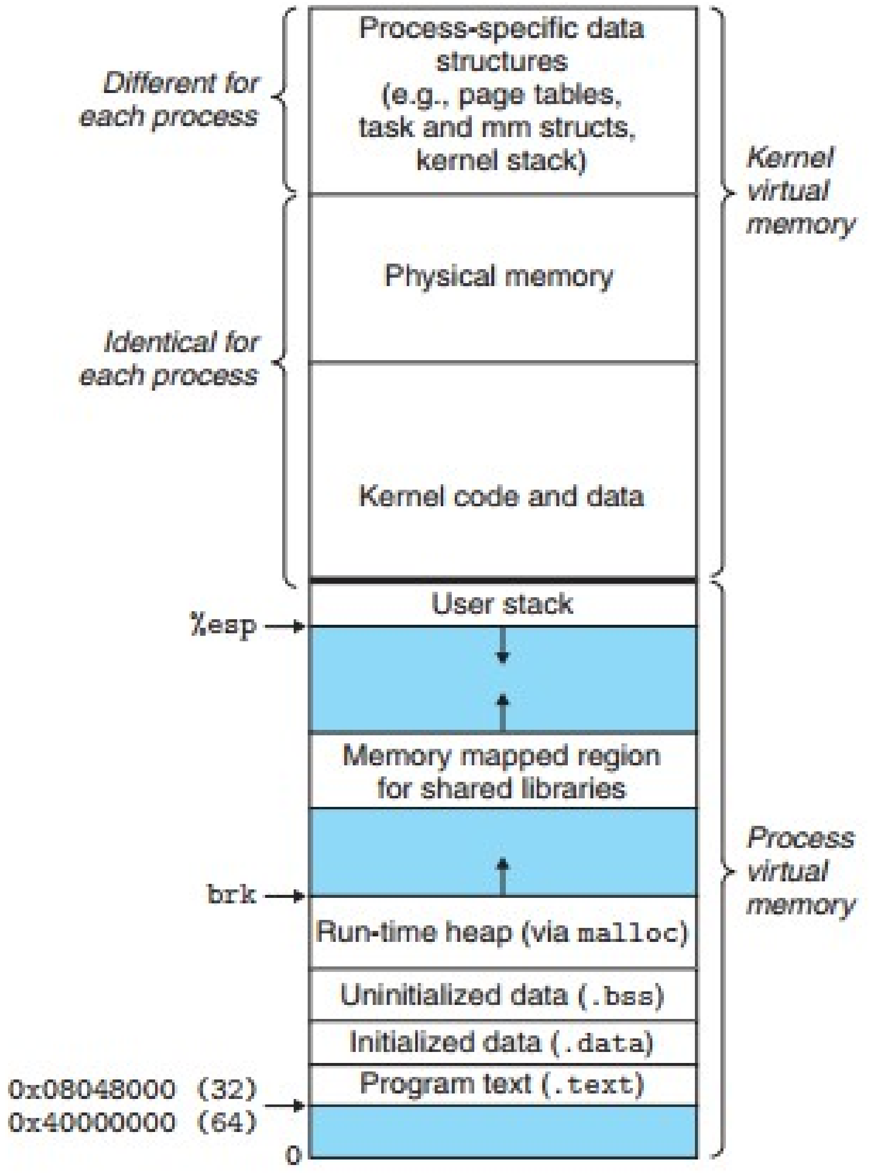
Virtual Memory

- The forbidden secret:
- We like to think of the virtual memory of a process as looking like this:



Virtual Memory

- It's actually more like this:
- The virtual memory space is split up into “user memory” which is the memory that is relevant to the running process.
- The “kernel memory” is memory that the OS/kernel requires.



Different for each process
Identical for each process
Process-specific data structures (e.g., page tables, task and mm structs, kernel stack)
Physical memory
Kernel code and data
Kernel virtual memory
User stack
%esp
Memory mapped region for shared libraries
brk
Run-time heap (via malloc)
Uninitialized data (.bss)
Initialized data (.data)
Program text (.text)
Process virtual memory
0x08048000 (32)
0x40000000 (64)
0

Virtual Memory

- For fun, let's define “kernel” now and pretend that we aren't 8 weeks late in doing so.
- The kernel is the main component of an operating system that essentially *does* all of the work of the OS.
  - Schedules the thread/process that is run by the CPU
  - Handles the virtual memory assignments of each process.
  - Deals with I/O
- The kernel is just another program

Virtual Memory

- A chunk of the virtual address space is reserved for the kernel's use (ex. storing the kernel's code)
- In some instances, for example, half of the space could belongs to the kernel.
- User space: 0x0000000000000000 – 0x7FFFFFFFFFFFFFFF
- Kernel space: 0x8000000000000000 – 0xFFFFFFFFFFFFFFFF

Virtual Memory

- This particular division gives the process about 2^{63} bytes of virtual memory space and the kernel 2^{63} bytes of virtual memory space.
- This is not enforced as some natural law or property. It's adjustable.
- In reality, because 2^{63} is 8 exabytes, there is no need to have so much memory.
- As a result we use 48-bit addresses.

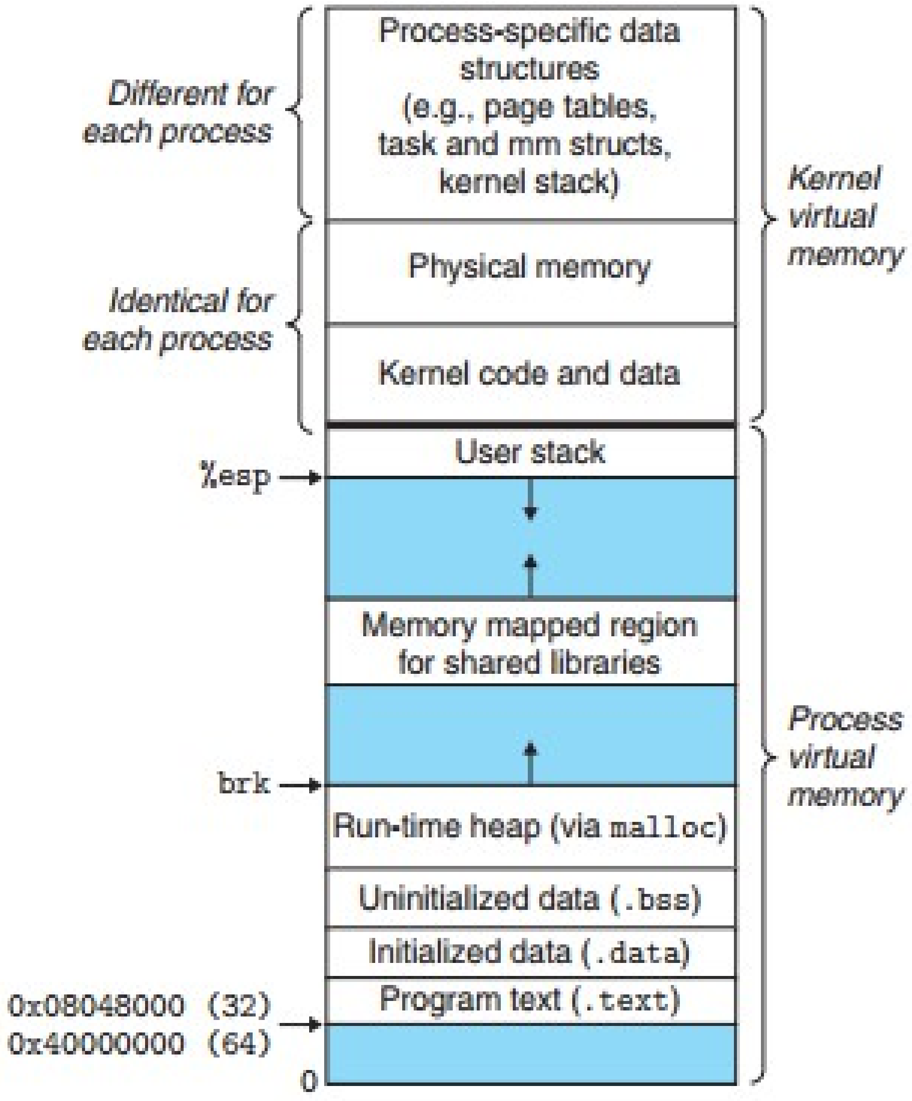
Virtual Memory

- The address is technically 64-bits but we can effectively implement 48-bit addresses by taking a 48-bit address and sign extending to 64-bits.
- Thus:
  - 0x7FFFFFFFFFFF becomes 0x00007FFFFFFFFFFF
  - 0x800000000000 becomes 0xFFFF800000000000
  - etc.
- In modern systems, user space is addresses 0 to 0x7F...FF and the kernel space is 0xFFF8000...000 to 0xFFF...FFF.
- Let's consider the book diagram again.

Virtual Memory

- CS:APP, pg. 829, the virtual memory of a Linux process:

Figure 9.26
The virtual memory of a Linux process.



Different for each process
Process-specific data structures (e.g., page tables, task and mm structs, kernel stack)
Physical memory
Identical for each process
Kernel code and data
Kernel virtual memory
User stack
%esp
Memory mapped region for shared libraries
brk
Run-time heap (via malloc)
Uninitialized data (.bss)
Initialized data (.data)
Program text (.text)
0x08048000 (32)
0x40000000 (64)
0
Process virtual memory

Virtual Memory

- CS:APP, pg. 829, the virtual memory of a Linux process:

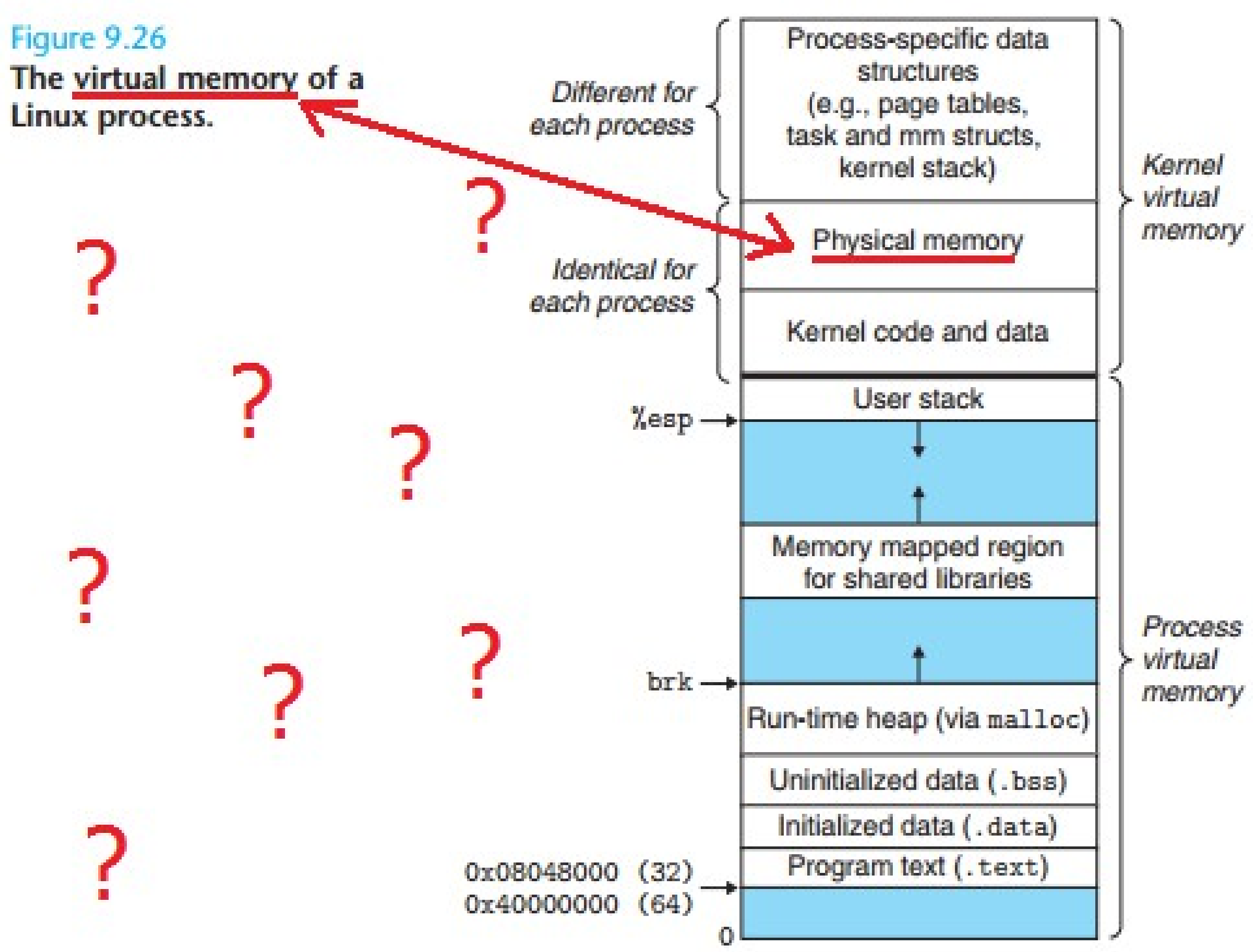


Figure 9.26
The virtual memory of a Linux process.
?
?
?
?
?
?
?
?
Different for each process
Identical for each process
Process-specific data structures (e.g., page tables, task and mm structs, kernel stack)
Physical memory
Kernel code and data
Kernel virtual memory
User stack
%esp
Memory mapped region for shared libraries
brk
Run-time heap (via malloc)
Uninitialized data (.bss)
Initialized data (.data)
Program text (.text)
Process virtual memory
0x08048000 (32)
0x40000000 (64)
0

Virtual Memory: User Space

- Unused buffer:
  - Addresses: 0x00000000-0x40000000
  - (Very generous) protection from dereferencing null or bad pointers.
  - You don't need to worry the next time you accidentally dereference 134000000, as we all tend to do.
- .text:
  - The source code of the loaded program.

Virtual Memory: User Space

- .data: initialized global variables, ex:

```
int a = 10; ←
int main()
{
  ...
}
```

- .bss: uninitialized global variables, ex:

```
int b; ←
int main()
{
  ...
}
```

Virtual Memory: User Space

- Why the distinction between bss and data?

Virtual Memory: User Space

- Why the distinction between bss and data?
  - When copying and loading the program from disk to be run, initialized global data must be copied from the disk to memory.
  - Uninitialized globals are assumed to be undefined, therefore we only need to allocate the necessary amount of space instead of having to read the file from disk.

Virtual Memory: User Space

- Heap:
  - For memory dynamically allocated via malloc
- Shared libraries:
  - The libraries included in dynamic linking (we'll get to that later)
- Stack
  - I really hope you know what the stack is at this point.

Virtual Memory: Kernel Space

- Kernel Code and Data
  - As mentioned above, the kernel is simply a program and therefore needs the same memory reservations to function correctly (.text, .data, etc).
  - This section is identical for each process (each process is running the same kernel)
- Here's where it's gets a bit weird. Yes. Here.

Virtual Memory: Kernel Space

- Process specific data
  - Resides at the uppermost sections of the memory.
  - The unique data and structures needed to maintain the correct execution of this process.
  - The book says this includes “page tables, mm structs, and the kernel stack”
  - The kernel stack: the kernel will have the same code for each process, but will be running different operations for each process, therefore, it needs its own stack
  - Makes sense

Virtual Memory: Kernel Space

- Process specific data
  - Page tables.
  - Wait, pages tables?
    - “Other regions of kernel virtual memory contain data that differs for each process. Examples include pages tables ...” (pg. 830, 3rd ed.)
  - So the page tables are actually stored in the kernel virtual memory?
    - “...and a data structure stored in physical memory known as the *page table* that...” (pg. 807 3rd ed.)
  - The page table is actually stored in main memory, but the process specific data of the kernel memory presumably maps to the page table in main memory.

Virtual Memory: Kernel Space

- Process specific data: task structs + mm structs

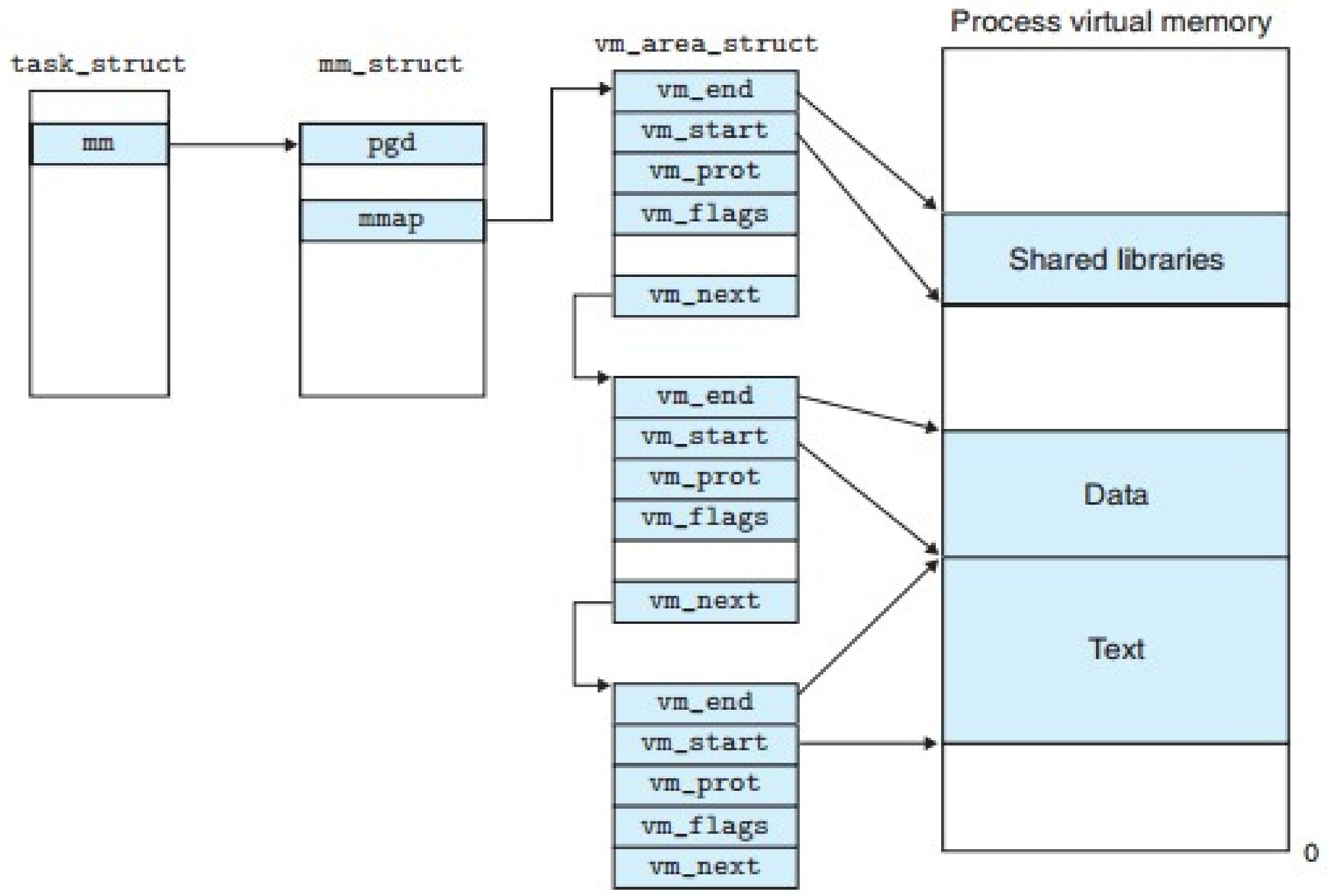


Figure 9.27 **How Linux organizes virtual memory.**

Virtual Memory: Kernel Space

- Process specific data: task structs + mm structs
  - task structs: holds info about the running process (such as pid, pointer to mm struct, and etc)
  - mm struct: holds info regarding the virtual memory of the process
  - mm struct holds the pgd: the level 1 page table of the process. When the process is run, this is stored into CR3, the page table register.
  - mm struct also holds mmap which points to vm_area_structs which maintain the used virtual memory.

Virtual Memory: Kernel Space

- Process specific data: task structs + mm structs
  - vm_area_struct is like a linked list
  - Each struct points to a range of allocated virtual memory (ex. stack, heap, etc.), as well as the next struct for the next range.

Virtual Memory: Kernel Space

- Finally, we get to this part:

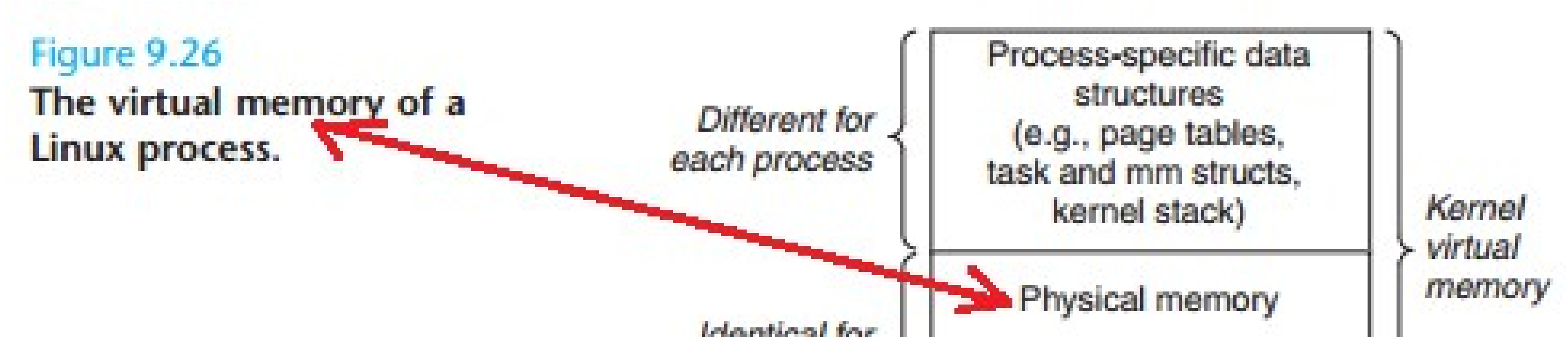


Figure 9.26
The virtual memory of a Linux process.
Different for each process
Process-specific data structures (e.g., page tables, task and mm structs, kernel stack)
Physical memory
Kernel virtual memory

- The virtual memory space contains physical memory?

Virtual Memory: Kernel Space

- We present the virtual addresses as being mapped to physical addresses via the page table.
- As in, a virtual address pointing to a page is mapped to a physical address where that page is actually located in physical memory.
- For example, if we take the virtual address that corresponds to the stack and translate that to a physical address, that address is the location of the stack in physical memory.

Virtual Memory: Kernel Space

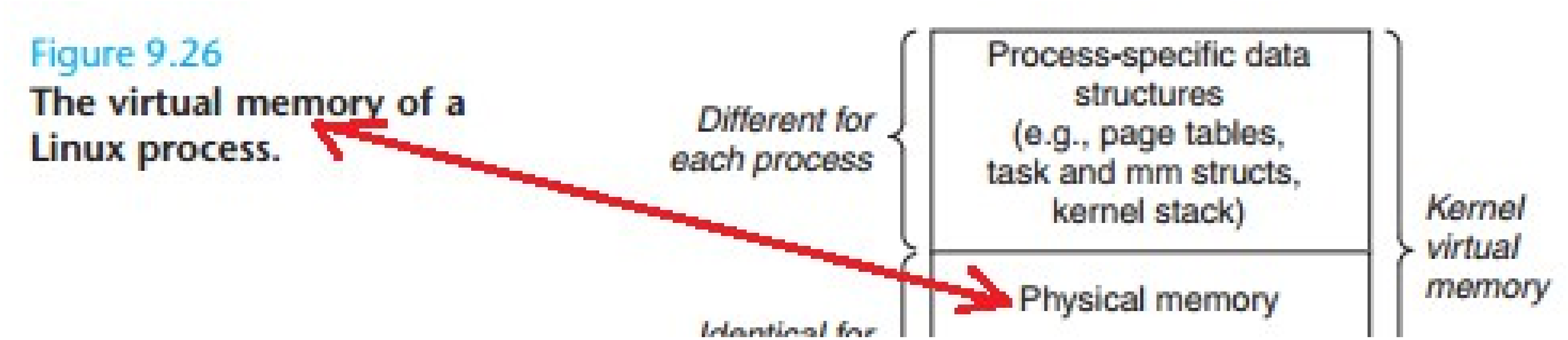


Figure 9.26
The virtual memory of a Linux process.
Different for each process
Process-specific data structures (e.g., page tables, task and mm structs, kernel stack)
Physical memory
Kernel virtual memory

- So we take the virtual address that corresponds to... physical memory... and then we get the physical address that corresponds to... physical memory.
- Book?

Virtual Memory: Kernel Space

- “Interestingly, Linux also maps a set of contiguous virtual pages (equal in size to the total amount of DRAM in the system (!?)) to the corresponding set of contiguous physical pages” (pg. 830 3rd ed.)
- So there's a range of virtual addresses that contain the entirety of the physical addresses?
- Short answer, yes.

Virtual Memory: Kernel Space

- “This provides the kernel with a convenient way to access any specific location in physical memory, for example, when it needs to access **page tables**, or to perform memory-mapped IO operations...”
- Of course. The page table again.

Virtual Memory: Kernel Space

- The kernel space uses the same TLB/page table mechanism to map virtual addresses to physical addresses.
- The kernel needs to be able to fool around with physical memory directly, but it cannot do so unless it has direct access to physical memory.
- As a result, the “physical memory” part of the kernel space simply translates virtual kernel addresses to physical addresses.

Virtual Memory: Kernel Space

- For example, if the “physical memory” of the kernel space begins at 0xFFFF8F0...000, then this might map to physical address 0x00000000. Then, 0xFFFF8F0...001 maps to 0x00000001 and etc.
- This physical memory part is shared among all of the processes.
- How feasible is this?

Virtual Memory: Kernel Space

- Consider 32-bit machines where 1 GiB is kernel space and 3 GiB is user space.
- We have 1024 MiB of kernel space, but the other components take up space.
- The number commonly thrown around on the internet is 128 MiB of other kernel usage.
- This means that the kernel space can only map 896 MiB of physical memory into the kernel space.

Virtual Memory: Kernel Space

- In order to map the rest of physical memory to virtual addresses, you can look up high memory vs. low memory kernel mapping.
- Be warned. If you start down this path, you may never see the light of day ever again.

Virtual Memory: Kernel Space

- Incidentally, consider 64-bit machines.
- Generally speaking, the user addressable space is:
- 0x000000000000 – 0x7FFFFFFFFFFF
- The kernel space is:
- 0xFFFF800000000000 – 0xFFFFFFFFFFFFFFFF
- This is 128 TiB. That's going to be enough to map the entirety of physical memory, at least for a while.

Virtual Memory: Kernel Space

- When you treat addresses as 48-bits, the virtual memory space seems contiguous.
- Highest User Address:
  - 0x7FFFFFFFFFFF
- Lowest Kernel Address:
  - 0x800000000000
  - 0x7FFFFFFFFFFF + 1 = 0x800000000000
- However, when you treat it as a 64-bit address:
- Highest User Address:
  - 0x00007FFFFFFFFFFF
- Lowest Kernel Address:
  - 0xFFFF800000000000
  - 0x00007FFFFFFFFFFF + 1 = 0x0000800000000000

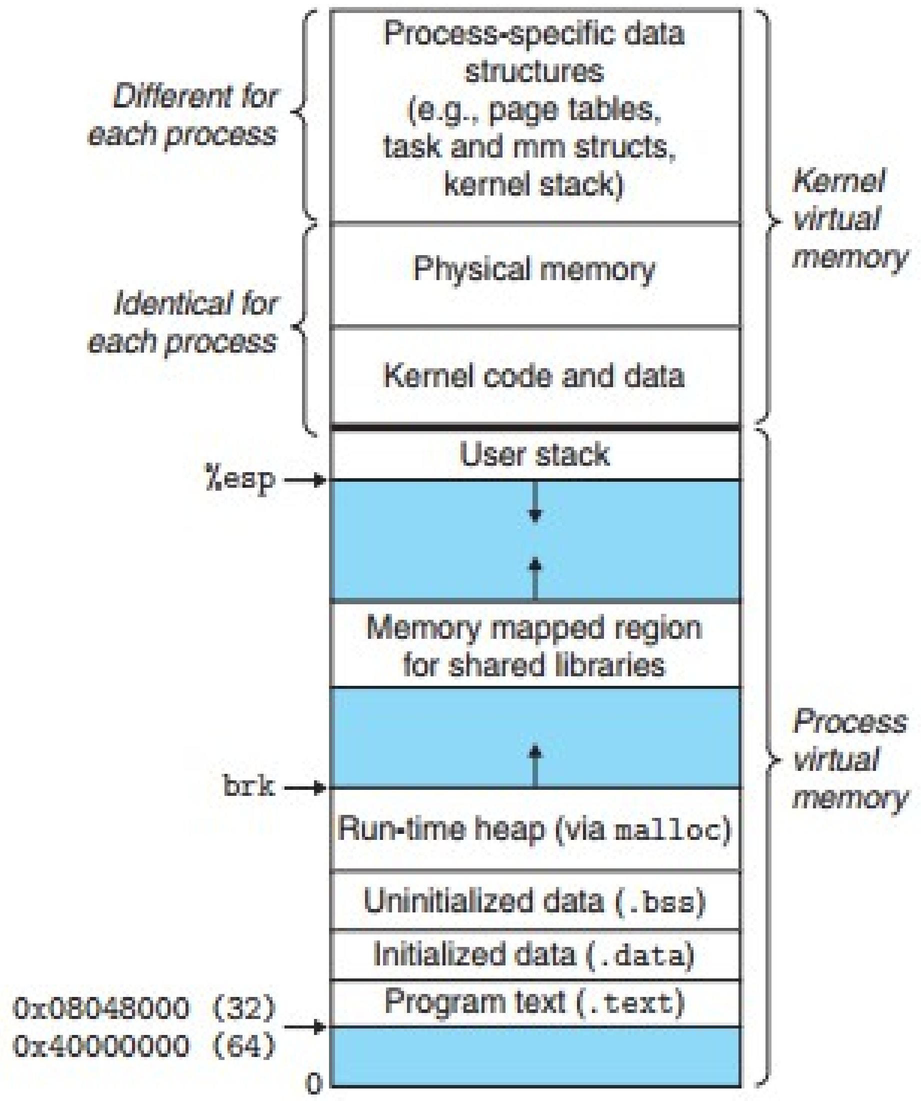
Virtual Memory: Kernel Space

- In typical current implementations, addresses where the leading 16 bits do not match the 47th bit:
  - Ex. 0x00008000....
- ...are invalid.
- As a result, the view of the virtual memory space is less like...

Virtual Memory

- ...this:
- ...and more like...

Figure 9.26
The virtual memory of a Linux process.


Process-specific data structures (e.g., page tables, task and mm structs, kernel stack)
Physical memory
Kernel code and data
Different for each process
Identical for each process
Kernel virtual memory
User stack
%esp
Memory mapped region for shared libraries
brk
Run-time heap (via malloc)
Uninitialized data (.bss)
Initialized data (.data)
Program text (.text)
0x08048000 (32)
0x40000000 (64)
0
Process virtual memory

Virtual Memory

- ...this:

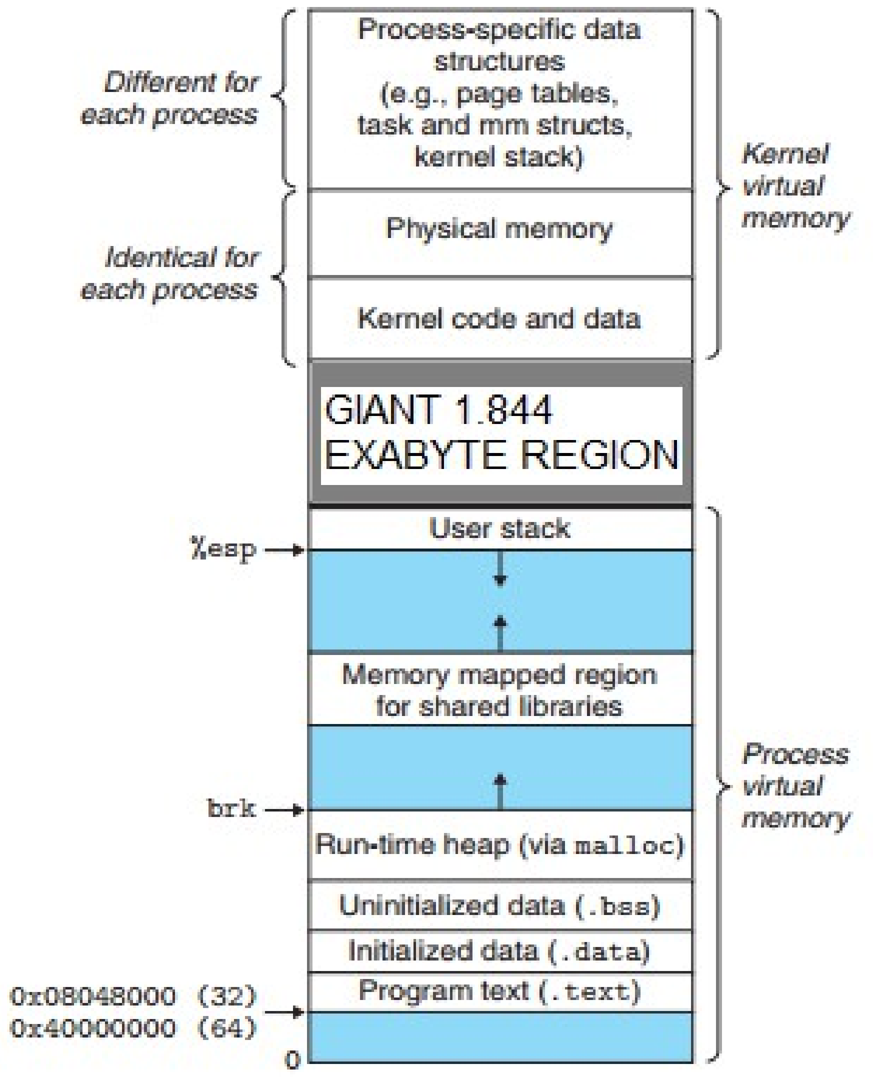

Different for each process
Identical for each process
Process-specific data structures (e.g., page tables, task and mm structs, kernel stack)
Physical memory
Kernel code and data
Kernel virtual memory
GIANT 1.844 EXABYTE REGION
User stack
%esp
Memory mapped region for shared libraries
brk
Run-time heap (via malloc)
Uninitialized data (.bss)
Initialized data (.data)
Program text (.text)
0x08048000 (32)
0x40000000 (64)
0
Process virtual memory

Figure 9.26
The virtual memory of a Linux process.

Linking

- Yup, it's this picture again:

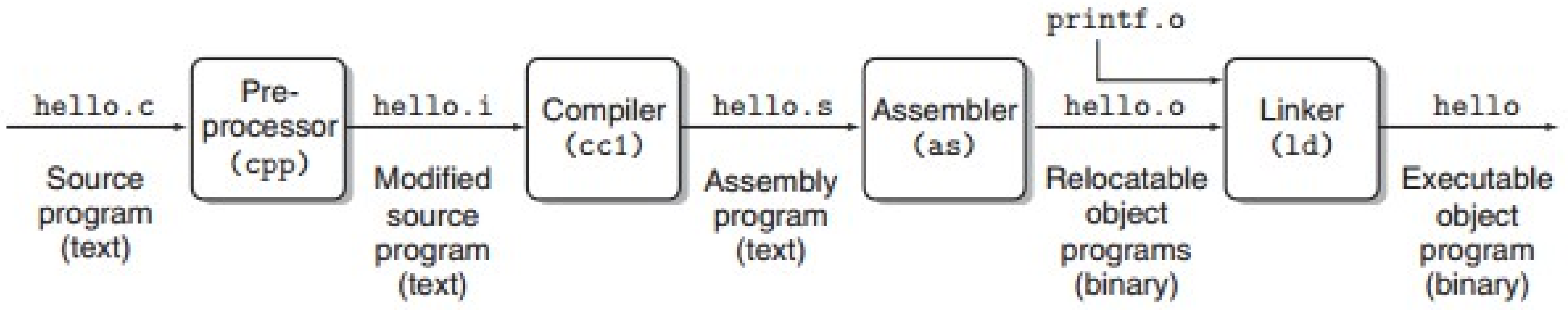


Figure 1.3 **The compilation system.**

Linking

- Except, we are here:

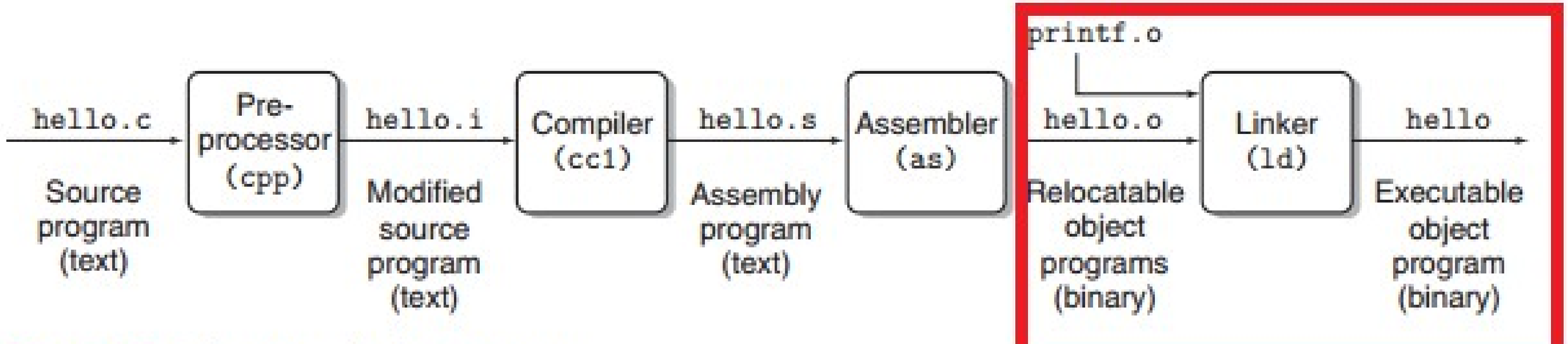


Figure 1.3 **The compilation system.**

Linking

1. Pre-processor: respond to and replace the compiler directives (lines marked with #) with the appropriate modifications

2. Compiler: Compile the C code into readable (so to speak) assembly.

3. Assembler: Compile the assembly into byte code/object files.

- Isn't that it? What else needs to be done?

Linking

- When something like this is done:
  - gcc main.c swap.c -o out
- ...the files for main.c and swap.c are compiled separately.
- Suppose we have the following:
- main.c:

```
int main()
{
  …
  swap();
}
```

swap.c:

```
void swap()
{
   ...
}
```

Linking

- main.c calls the function “swap” that is defined in swap.c
- main.c is compiled independently of swap.c, how does it know what swap is?
  - We need to do symbol resolution.
- Function calls look like “call 0x400688”. How do we know the address of swap? What is even the address of swap at this point?
  - We need relocation.

Static Linking

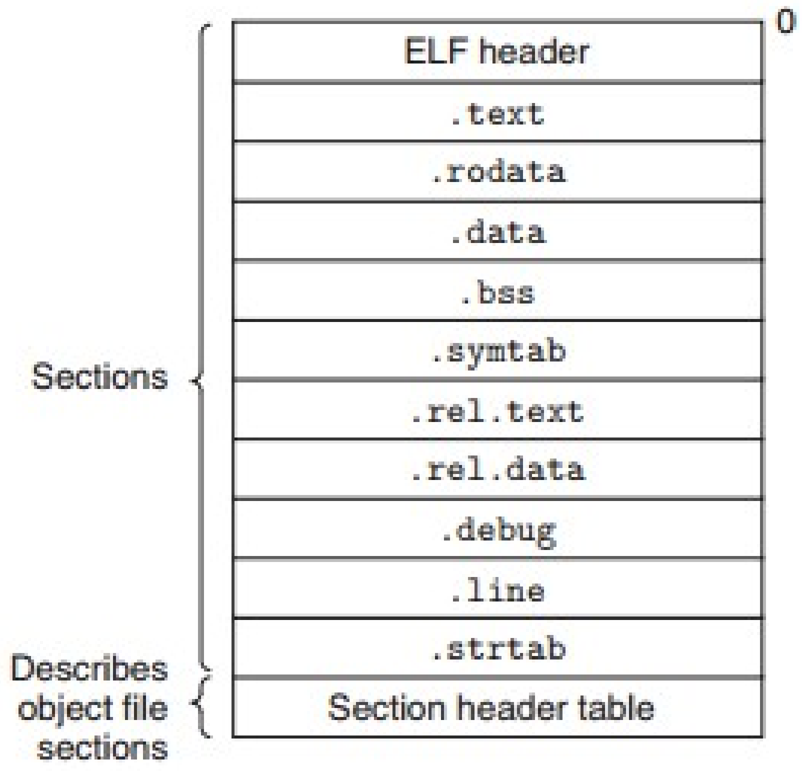
- Because of the issues described on the previous page, it's clear that there's a little more to compilation than simply translating C to assembly.
- Eventually, the entirety of the (statically linked) code will be in the .text section where all of the code will exist in the same place.
- However, because files are assembled separately, there is no prior knowledge of their position in memory.

Static Linking

- The reality is that the assembler step produces “relocatable object files”.
- These relocatable object files cannot be executed. Instead, they are primed so that they can be included as part of a project later on.
- When the relocatable object files are linked together, they form an “executable object file”, the final executable.

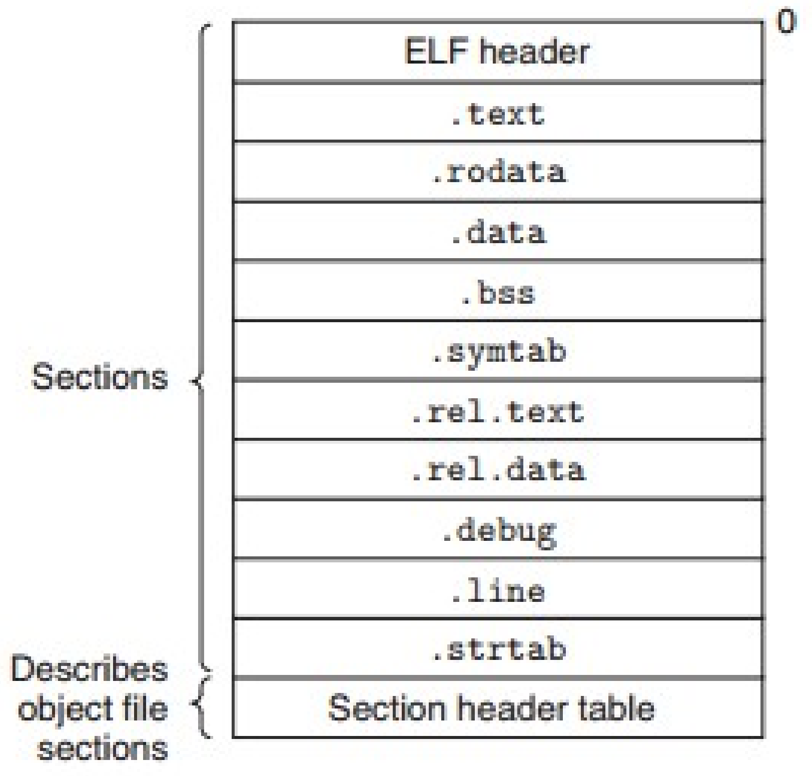
Static Linking

- Relocatable object file format:
- ELF header
  - File info (word size, byte ordering)
- .data, .bss
  - Same as in full object file



Static Linking

- .text:
  - The code of the compiled program
  - Incomplete (symbols are not resolved, relocation is not done)
  - In place of actual addresses for functions/variables that must be relocated, placeholder stubs are present.
- .rodata
  - “Read only data”
  - Things like jump tables, printf strings.



Static Linking

- Placeholder?
  main.c:
```
#include <stdio.h>
int main()
{
  foo();
}
```

- gcc -m32 -c main.c

- “-c” stops compilation after the assembler and before the linker.

Static Linking

```
Disassembly of section .text:
00000000 <main>:
    0:   8d 4c 24 04                  lea    0x4(%esp),%ecx
    4:   83 e4 f0                     and    $0xfffffff0,%esp
    7:   ff 71 fc                     pushl  -0x4(%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>
   16:   83 c4 04                     add    $0x4,%esp
   19:   59                           pop    %ecx
   1a:   5d                           pop    %ebp
   1b:   8d 61 fc                     lea    -0x4(%ecx),%esp
   1e:   c3                           ret
```

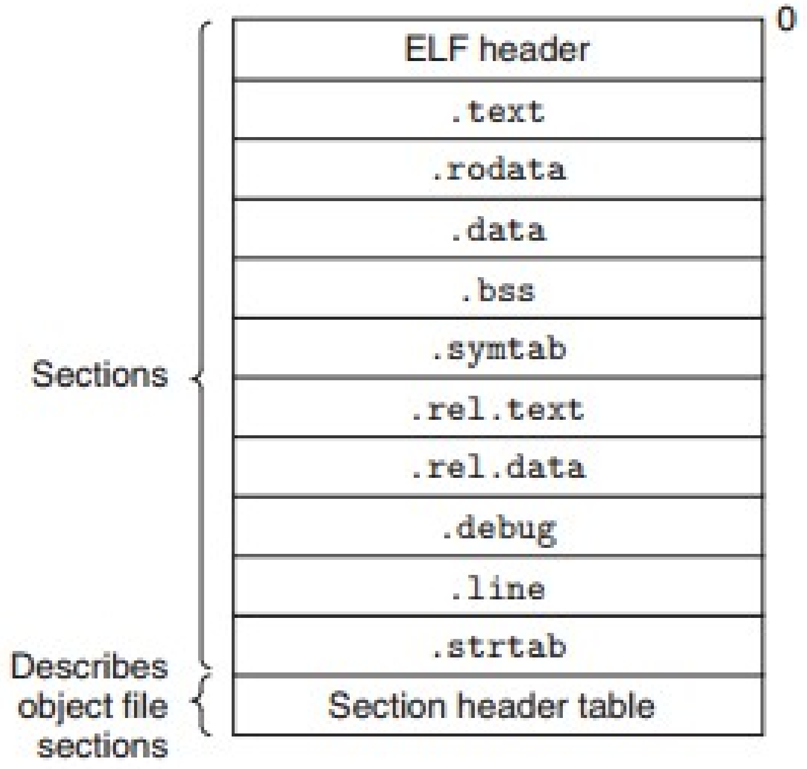
- The call to “foo” is initially a call to the current address + 1. This points to fc ff ff ff or -4 when the endianness is accounted for.

Static Linking

- This requires:
  - Symbol resolution: associate “foo” with the function that is to be called.
  - Relocation: find the address of the “foo” function and replace the stub.
- How does the linker know that this call needs to be replaced?

Static Linking

- .rel.text:
  - "relocation text"
  - List of locations in the .text section that will need to be modified to contain the actual addresses of functions/data.



Static Linking

- What about for data?
- main.c:

```
#include <stdio.h>
extern int blah;
int main()
{
  int i = 0x12345678 + blah;
}
```

- The extern keyword means that the global variable blah is explicitly defined in another module.

Static Linking

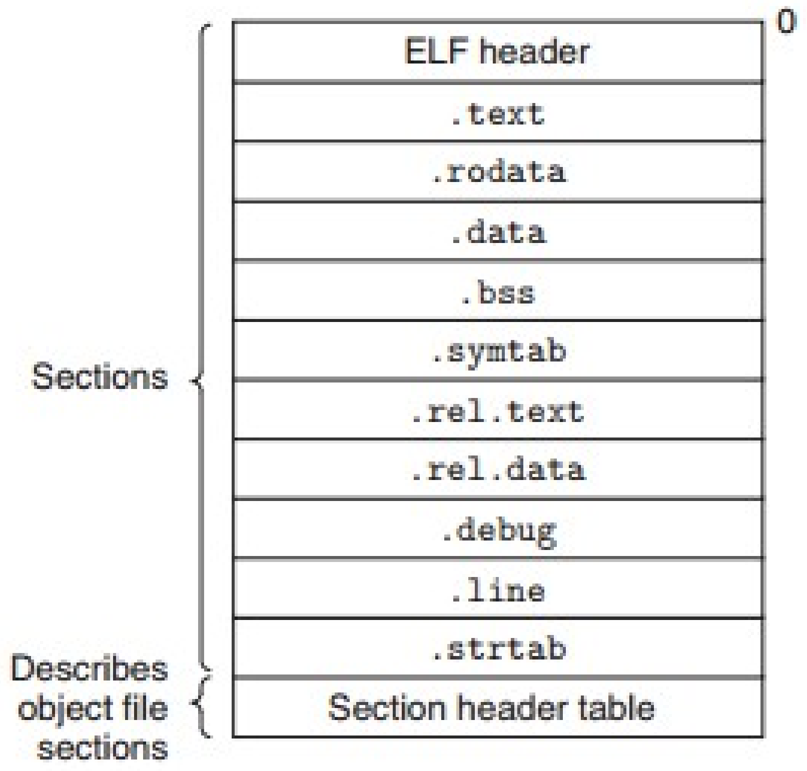
```
Disassembly of section .text:

00000000 <main>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   83 ec 10                sub    $0x10,%esp
 → 6:   a1 00 00 00 00          mov    0x0,%eax
   b:   05 78 56 34 12          add    $0x12345678,%eax
  10:   89 45 fc                mov    %eax,-0x4(%ebp)
  13:   c9                      leave
  14:   c3                      ret
```

- The placeholder is “0” in anticipation of the actual global variable.
- Eventually, this mov will load from a memory address that is the address of the actual variable.

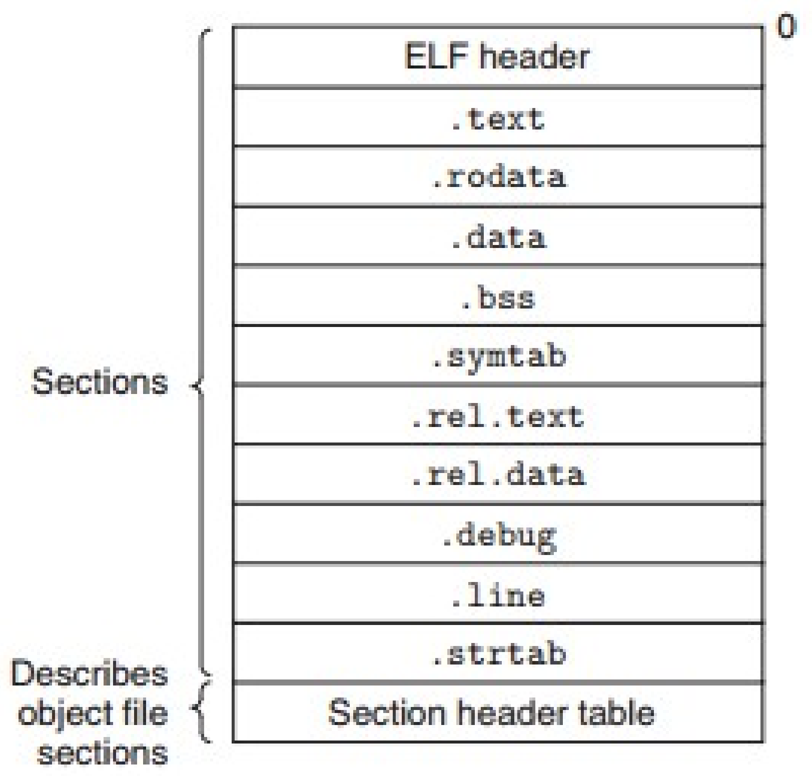
Static Linking

- .rel.data:
  - “relocation data”
  - Information about global variable referenced or defined in this module.



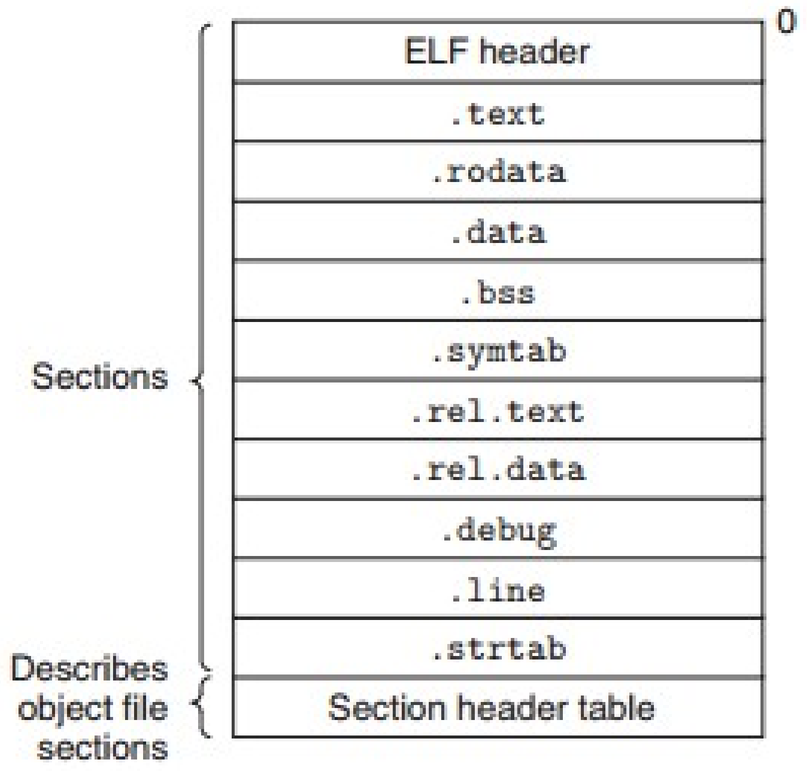
Static Linking

- .symtab:
  - "symbol table"
  - List of symbols that are defined and referenced in this module
  - Ex. names of functions, global/static variables, etc.
  - Does NOT contain symbols of local variables



Static Linking

- .strtab:
  - A string table that contains the actual strings of the symbols referred to in the symbol table.



Symbols

- The three types of symbols:
- 1. Global symbols defined by module m that can be referenced by other modules:
  - non-static global variables and functions

Symbols

- The three types of symbols:
- 1. Global symbols defined by module m that can be referenced by other modules:

foo.c:

```
extern int z;
int fun() ←
{
  int y = 10;
  z = 10;
}
```

bar.c:

```
static int v;
int z;
static void blah()
{
  v = 0;
  fun();
}
```

Symbols

- The three types of symbols:
- 2. Global symbols referenced by the module but defined by another module.
  - Ex: externals

Symbols

- The three types of symbols:
- 2. Global symbols referenced by the module but defined by another module.

foo.c

```
extern int z; ←
int fun()
{
  static int x = 0;
  int y = 10;
  z = 10; ←
}
```

bar.c

```
static int v;
int z;
static void blah()
{
  v = 0;
  fun();
}
```

Symbols

- The three types of symbols:
- 3. Local symbols that are defined but used exclusively in this module.
  - Functions and global variables that are defined by static.
  - Recall, when static is applied to globals/functions, that means this variable/function is only visible to THIS module.
  - These “local” symbols do NOT refer to local variables. Non-static local variables DON'T need representation in the symbol table.

Symbols

- The three types of symbols:
- 3. Local symbols that are defined but used exclusively in this module.

foo.c:

```
extern int z;

int fun()

{

  static int x = 0;

  int y = 10;

  z = 10;

}
```

bar.c:

```
static int v; ←
int z;
static void blah() ←
{
  v = 0;
}
```

Note on static local variables

- A note regarding static:

```
int f()
{
  static int x = 0;
  return x;
}
```

```
int g()
{
  static int x;
  return x;
}
```

- Are these two x's the same variable?

Note on static local variables

- When we say there is only one instance of a static local variable, this means that there is one instance of a static local as defined by that function.
- Thus, every time we call f(), we will be referring to the static int x defined by f (let's call this f.x). Any threads calling f() will find that they operate on the same x.
- However, the x defined by g() is distinct value (let's call this g.x). Any calls to g() will be referring to the same g.x, which is distinct from f.x.

Note on static local variables

- Static local variables are stored in the .data section.
- In this example f.x and g.x will both have some spot in the data or bss section.
- Additionally, they will require symbols in the symbol table for resolution purposes.
- These fall under case 3, “local symbols”.

Symbol Resolution

- When it comes time to link, each unknown symbol will be resolved to one entry of one of the symbol tables of the input relocatable object files.
- Local symbols are easy to resolve since the symbols in the code are only referenced in the same module.
- Globals are trickier since there can be multiple global variables with the same name.

Symbol Resolution

- Each global symbol is considered to be “strong” or “weak”. This helps to resolve symbols when multiple symbols have the same name:
- Rule 1: Multiple strong symbols not allowed
- Rule 2: If there is a strong symbol and multiple weak symbols, choose the definition of the strong symbol.
- Rule 3: If there are multiple weak symbols, choose the weak symbol with the largest data type.

Symbol Resolution

- Functions are strong symbols (no two functions in the set of modules can have the same function name)
- Initialized global variables (int glob = 10;) are strong symbols.
- Uninitialized global variables are weak symbols.

Symbol Resolution

- POST DISCUSSION UPDATE:
  - You CAN (ie the linker will allow it) define a function name (a strong symbol) in one module and an uninitialized variable with the same name (weak symbol) in another module. Ex. int foo(){...} in one module and in foo; in another. Since the strong symbol wins, if you try to do foo = 10, you attempt to write “10” to the location of the foo function, which segfaults.
  - You CAN define a function (a strong symbol) in one module and a static function with the same name (a ??? symbol?) in another module. The module with the static function will just call the static version.
- PLEASE DON'T DO EITHER OF THESE IN REAL CODE

Symbol Resolution

- Consider main.c and bar.c

main.c:

```
int x;
int main()
{
  x = 10;
  f();
  printf("%d\n", x);
}
```

bar.c:

```
int x;
void f()
{
  x = 20;
}
```

Symbol Resolution

- Both x's are weak symbols. Thus, they are resolved to be the same object.
- main will initialize the x to be 10. Then (perhaps unknowingly), f() will assign 20 to x.
- This can be a problem if different people are working on different modules and both happened to name global variables “x”.
- Assume Person A is working on main.c and Person B is working on bar.c

Symbol Resolution

- If both individuals don't know that x is also defined elsewhere, then they may think that they're working on a global variable that only exists in the particular module.
- Then, the value of x seems to change arbitrarily.
- This can be an even more confusing problem if two declarations of x are of a different type.

Symbol Resolution

- Prevent this with the gcc flag “-fno-common” which makes all symbols strong, disallowing any multiple definitions.
- “-fno-common”
  - Professor Eggert approved.
- This technique is called “common storage allocation”, hence “-fno-common” to disable.

Relocation

- Once symbol resolution is done, each symbol referenced in code is associated with a single symbol table entry defined in one of the .o (relocatable object) files.
- Now, we need to copy all of the code into one place.
- This is going to rearrange the locations of all of modules.
- For each reference to a symbol, now we need to associate it with an address.

Relocation

- Two steps:
- Merge the sections:
  - All of the .data sections of the .o files are merged into one. All of the .text sections are merged into one, etc.
  - Each instruction and global variable will have a unique address.
- Relocate symbol references:
  - Change every symbol reference so that it points to the right location.

Relocation

- For each reference to a symbol that needs to be relocated, there will be an associated relocation entry (either .rel.text or .rel.data)
- Each entry contains:
  - offset : the offset in this module to the line that requires relocation
  - type: the type of relocation
  - symbol: the index into the symbol table of the symbol that this should be relocated with
  - addend: an offset to an address (more on that later).

Relocation

- There are two common forms of relocation:
- R_386_PC32: PC relative relocation
  - The relocation is based on an offset +/- the current %eip value.
- R_386_32: Absolute addressing
  - The relocation is based on the exact address.
- The book goes into more detail than the professor so you probably don't need to know it too much in depth, but here's the quick and easy.

Relocation

- Consider a main.c and sum.c that respectively define the main and sum functions. main.c is the following:

  main.c:

  ```
  int array[2] = {1, 2};
  int main()
  {
    int val = sum(array, 2);
    return val;
  }
  ```
- What needs to be relocated here?

Relocation

- Consider a main.c and sum.c that respectively define the main and sum functions. main.c is the following:

  main.c:
  ```
  int array[2] = {1, 2};
  int main()
  {
    int val = sum(array, 2);
    return val;
  }
  ```
- What needs to be relocated here? array and sum.

Relocation

- <main>:

```
0:  48 83 ec 08      sub   $0x8, %rsp
4:  be 02 00 00 00   mov   $0x2, %esi
9:  bf 00 00 00 00   mov   $0x0, %edi    #%edi = &array
        a: R_X86_64_32  (absolute relocate, array)

e:  e8 00 00 00 00   callq 13<main+0x13> #sum()
        f: R_X86_64_PC32 (relative relocate, sum)
13: 48 83 c4 08      add   $0x8, %rsp
17: c3               retq
```

Relocation

- Let's consider the relative relocation example.

```
e:  e8 00 00 00 00  callq 13<main+0x13> #sum()
            f: R_X86_64_PC32 (relative relocate, sum)
```

- The relocation entry will an offset of 0xF because the relocation necessary is 15 addresses away from the beginning of the module.
- The “symbol” of the relocation entry will contain the index of the symbol table. When linking, the main function and sum function will be placed into memory.
- Say they are placed into memory as follows:

Relocation

```
main:
  4004d0:  48 83 ec 08     sub   $0x8, %rsp
  4004d4:  be 02 00 00 00  mov   $0x2, %esi
  4004d9:  bf ?? ?? ?? ??  mov   ??, %edi #&array
  4004de:  e8 ?? ?? ?? ??  callq ?? <sum>
  4004e3:  48 83 c4 08     add   $0x8, %rsp
  4004e7:  c3              retq
sum:
  4004e8:  b8 00 00 00 00  mov   $0x0, %eax
  4004ed:  ba 00 00 00 00  mov   $0x0, %edx
  ...
```

Relocation

- main and sum have been positioned, and now main needs replace its stub call to sum with the actual call to sum.
- With PC (program counter) relative addressing, this is done by finding the offset to sum relative to the %rip value of the call to sum.
- More specifically, we need to identify the offset from the address of the NEXT instruction after the call and the address of sum.

Relocation

- In the example, we have:

```
4004de:   e8 ?? ?? ?? ??   callq ?? <sum>
4004e3:   48 83 c4 08      add   $0x8, %rsp
```

- The next instruction is at 0x4004e3. We also have:

- `sum:`

```
4004e8:   b8 00 00 00 00   mov   $0x0, %eax
4004ed:   ba 00 00 00 00   mov   $0x0, %edx
```

- The sum is at 0x4004e8

Relocation

- The difference between the next instruction and sum is 0x4004e3 – 0x4004e8 = 5.
- Thus, we use 5 to fill in the instruction:

```
4004de:  e8 ?? ?? ?? ??  callq ?? <sum>
```

- ..becomes...

```
4004de:  e8 05 00 00 00  callq 4004e8 <sum>
```

- Note the little endian order.
- This computation is accomplished through the use of the relocated locations of main and sum, the offset of the necessary relocation, etc.

Relocation

- The actual computation is more complication than this and can differ slightly between machines, but this is the gist of it (when it comes to x86-64 anyway).
- If you'd like to know more, it's on pg. 693 (3rd ed.).

Relocation

- R_386_32: Absolute addressing:
- Consider:

main.c:
```
int blah;
int main()
{
   ...
}
```

blah.c:
```
int *ptr = &blah;
int func()
{
   ...
}
```

Relocation

- Say we know that ptr is in the data section here:

```
0804945c <ptr>:
  0804945c: XX XX XX XX
```

- If we know that blah is at 0x8049454, then we replace the value of ptr with:

```
0804945c <ptr>:
  0804945c: 54 94 04 08
```

- Now, the value of ptr is &blah.

Static Linking

- Static linking combines all of the .o (relocatable object files) and compiles them into one.
- The resulting executable contains a combination of all of the code defined in the .o files.
- However, how do we statically link standard libraries functions?
- Consider a program that uses printf, scanf, etc.

Static Linking

- 1. Include all of the .o files of the functions you need to use from the standard library.
  - gcc main.c /usr/lib/printf.o /usr/lib/scanf.o /usr/lib/fgetc.o /usr/lib/fgets.o /usr/lib/fputs.o /usr/lib/putchar.o /usr/lib/ohi.o /usr/lib/karen.o /usr/lib/hell.o /usr/lib/jell.o /usr/lib/scottbai.o /usr/lib/may.o /usr/lib/hey.o /usr/lib/yokoon.o /usr/lib/supermari.o …
- This works, but it's a hell of a thing to compile.
- Long and error prone.

Static Linking

- 2. Include one file that includes every standard library function.
- gcc main.c /usr/lib/libc.o
- This also works, but this means that each executable has it's own copy of the entire library, which makes the executable very large.

Static Linking

- The solution to this is an .a (archive) file.
- An archive file contains names/references to all of the .o files defined by the library. However, upon compilation, the linker will scan through your program and decide which files it needs.
- Then, it will only copy over the .o files referenced by the .a file that are needed by your program.

Static Linking

- This process is a little nuanced and in fact the order of files matters. Say main.c uses a function f defined in f.o in archive.a.
- gcc main.c archive.a
- This works fine. main is scanned first and the linker remembers that f must be resolved. Then, archive a is scanned and since it has a reference to f.o, it can resolve the function f in main.

Static Linking

- However, if we have:
- gcc archive.a main.c
- archive.a is scanned first. At this point, the linker doesn't know any symbols that must be resolved.
- Then, main.c is scanned and the compiler notes that “f” must be resolved.
- But now... we've scanned all of the files and we have an unresolved “f”. This causes an error.

Static Linking

- The rule of thumb is to put the archive files last.
- What if you have a circular reference?
  - main.c uses “f”, which is defined in libx.a
  - f.o in libx.a uses a function g defined in liby.a
  - g.o in liby.a uses a function h defined in libx.a
- gcc main.c libx.a liby.a ?

Static Linking

- gcc main.c libx.a liby.a
- When we scan main.c, we recognize that we need f.o.
- When we scan libx.a, we resolve f.o and we need g.o.
- When we scan liby.a, we resolve g.o and we need h.o.
- We've got an unresolved symbol now.

Static Linking

- Solution:
- gcc main.c libx.a liby.a libx.a
- When we scan main.c, we need f.o.
- When we scan libx.a, we resolve f.o and we need g.o.
- When we scan liby.a, we resolve g.o and we need h.o.
- When we scan libx.a, we resolve h.o

Dynamic Linking

- Static linking means that when the program is compiled, all of the code necessary to run is packaged in the final executable file.
- Why is this a nice property?

Dynamic Linking

- Static linking means that when the program is compiled, all of the code necessary to run is packaged in the final executable file.
- Why is this a nice property?
  - The entire program is in one location, no other dependencies.
  - The cost of linking is entirely done during compile time. Execution and loading is fast.
- In what way is this undesirable?

Dynamic Linking

- In what way is static linking undesirable?
  - Duplicate code: Each program contains a copy of the standard library functions it needs to use. This means a lot of duplicate code among different programs. If you have multiple programs running at the same time, you'll have multiple copies in memory, wasting space.
  - If you need to update the standard library, you have to update every executable.
- Enter: dynamic linking.

Dynamic Linking

- Dynamically linkable libraries are (.so) files, or shared libraries. Like normal object files, these contain text and data that need to be relocated.
- These are shared in the sense that there is one of these .so files for all of the executables on a machine.
- Instead of each executable having a copy of the linked code in the .text section during compile time, we'll leave some placeholders.
- During run time, dynamically link the shared libraries.

Dynamic Linking

- With static linking, each executable has a copy of library code in physical memory when running.
- With dynamic linking, the .so is copied into physical memory and shared among all of the running process via virtual memory sharing (different processes map different virtual addresses to the same physical address).
- Only one copy of the library at any point in time.

Dynamic Linking

- Downsides?
  - Cost of linking is moved from compile time to run time.
  - A single faulty shared library will break everything.

Dynamic Linking

- So how can this be accomplished?
- A simple approach
  - If a function calls a function defined by a shared library, replace the function call with a stub.
  - At load (or run time), pull the library code/data into virtual/physical memory as necessary.
  - Use the relocation/symbol resolution techniques to alter the stubs to point to the correct location.
- This approach works for when user defined text needs to reference a shared library, but what about the case where the shared library needs to reference another shared library?

Dynamic Linking

- Consider the following situation. We dynamically link two shared libraries:
  - shared1.so
  - shared2.so
- shared2.so defines function “fun2” in its text section.
- shared1.so uses the function “fun2” which is defined by shared2.so

Dynamic Linking

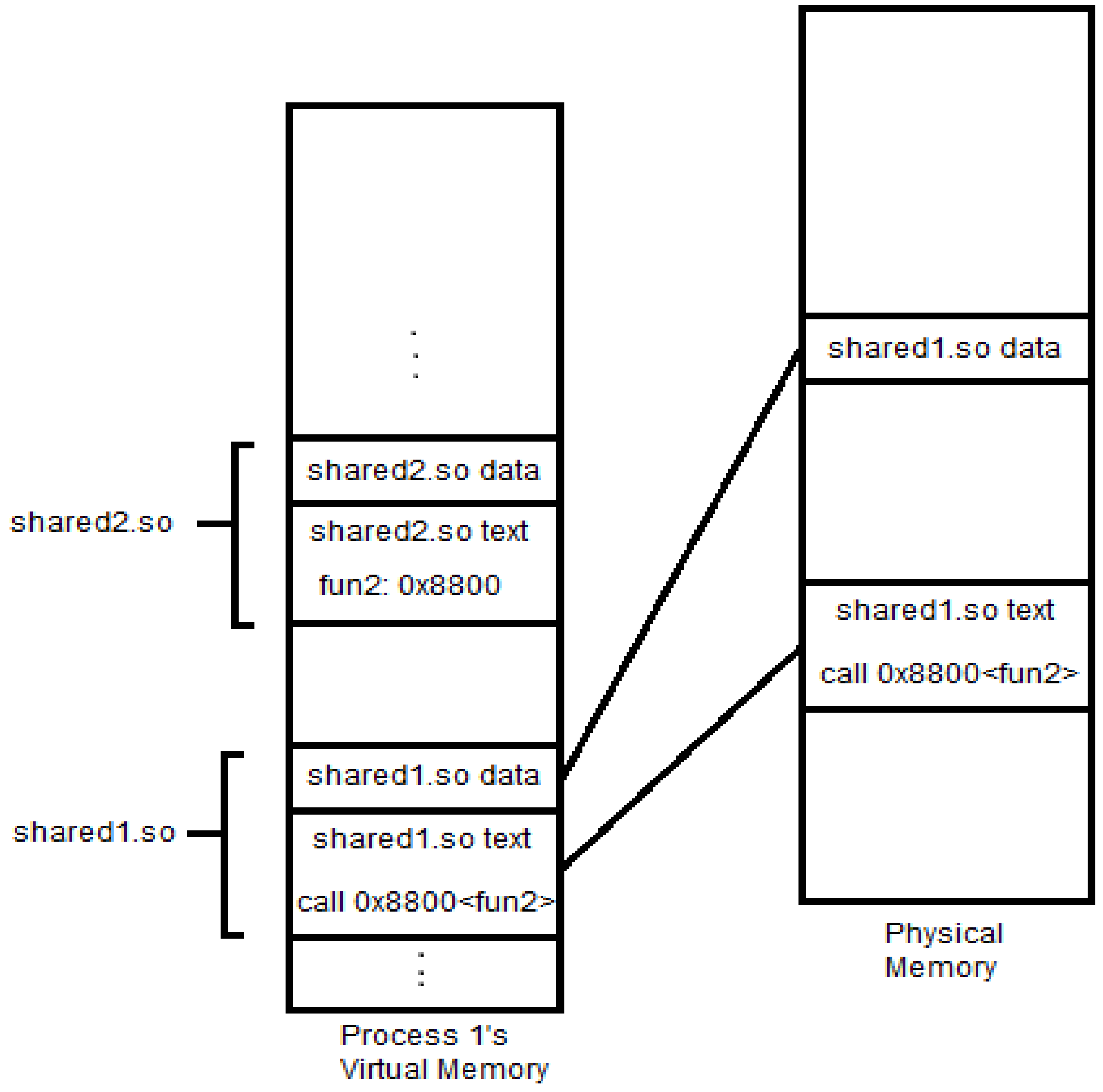
- The simple approach:
  - As we load shared1.so and shared2.so, we need to relocate shared1.so's reference to shared2.so's “fun2”.
  - The references to “fun2” in shared1.so must now be changed using the previous symbol resolution and relocation methods.
- Some observations:
  - Each of the shared library's data section should NOT be shared because global variables should be process specific.
  - Our goal is to share the text of the shared libraries.

Dynamic Linking

- Assume the following:
- shared1.so is at address 0x8000.
- shared1.so contains the line:
  - call fun2
- fun2 is located in shared2.so's text section and is at address 0x8800.
- Thus, the line in shared1.so is changed from:
  - call XX<fun2>
- to
  - call 0x8800<fun2>

Dynamic Linking

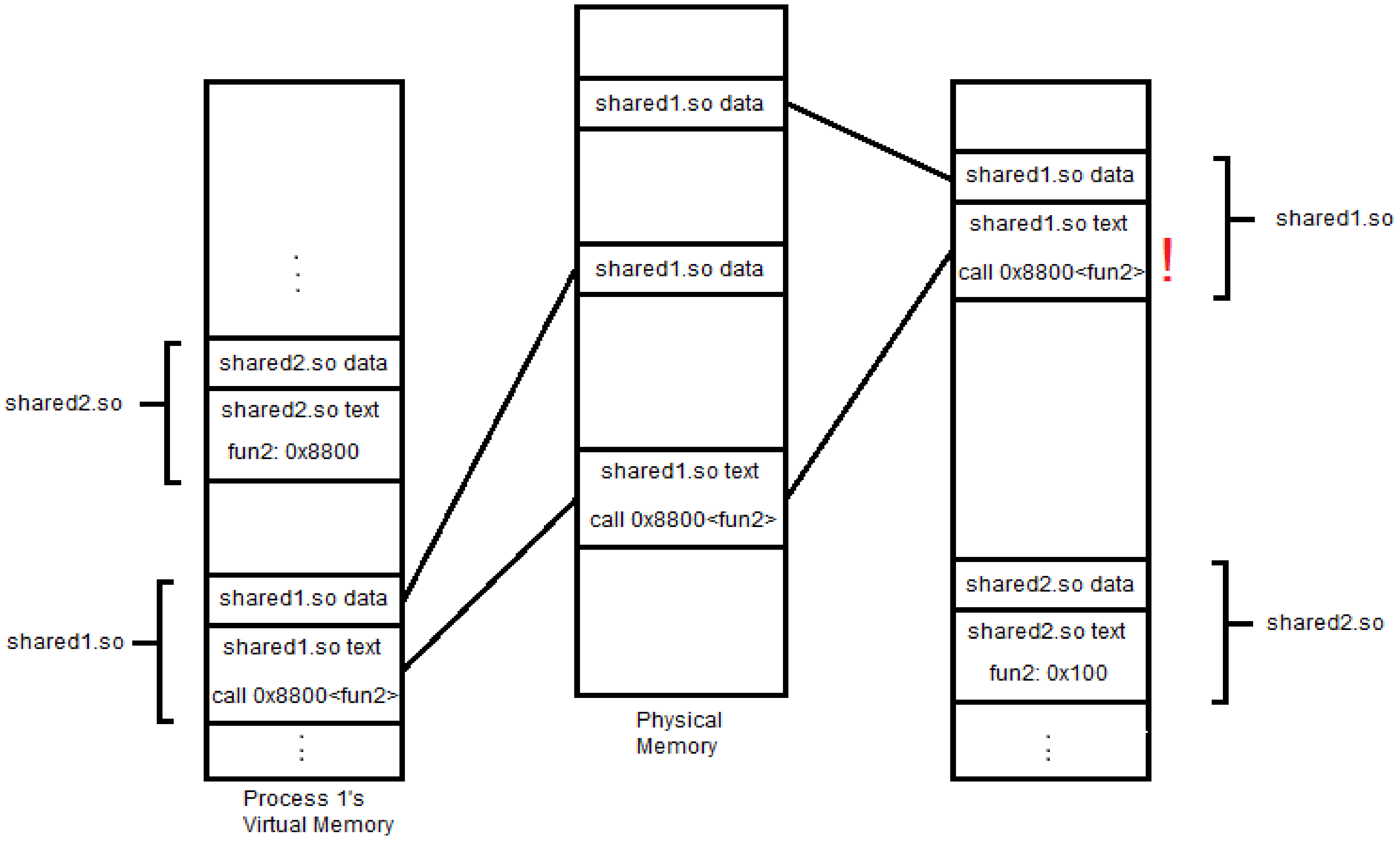
- This may look something like this:



Dynamic Linking

- Now let's try to link this shared library into another process. shared1.so's data shouldn't be shared between the two, but shared1.so's text SHOULD be.
- Note, it would be an unfeasible restriction to assume that process 2 also links the library into the same virtual address.
- Let's assume that process 2 links shared1.so into virtual address 0x100000. Meanwhile, the fun2 in shared2.so is linked at address 0x100

Dynamic Linking



Dynamic Linking

- Using this method of relocation, we replaced the stub call to fun2 in shared1.so's text with the virtual address of fun2, relative to the first process.
- However, now as we try to share shared1.so's text, we also get the same address of fun2, which is NOT accurate for the first process.
- In order for this to work, each .so library would have to be loaded into a fixed location, which would be impractical given the number of libraries we would want to share and the different processes that we would want to share the library with.

Dynamic Linking

- The solution is to use Position Independent Code, that is to say, the text segment (which should be shared) is position independent and has no addresses relative to a specific process' virtual memory.
- However, the shared1.so file includes both a text segment and data segment. The data section is assumed to be NOT shared.
- Position Independent Code works by moving all process unique addresses to the non-shared data section.

Dynamic Linking

- The observation that we make is that regardless of where we include shared1.so into process 1 or 2, the distance between the shared1.text and shared1.data is the same.
  - Ex. if a library's text segment begins at 0x100 and the data segment begins at 0x200, then if the library's text segment was loaded into 0x8000 then the data segment would be at 0x8100.
- The data segments will be mapped to different physical addresses, but this offset is consistent in the virtual address space.

Dynamic Linking

- Thus, we store a “Global Offset Table” in the data section, whose values are unique for each process.
- There will be an entry into the Global Offset Table (GOT) for each global variable reference. The address of that reference is the GOT entry.
- At a high level, fun2's address is stored in a GOT in the shared1.so's data section.
- Each process will have a different shared1.so data section so the GOT entry will be different among processes.

Dynamic Linking

- Now, instead of calling an absolute address, we call an indirect address, which is the Global Offset Table entry for fun2.
- The GOT entry for fun2 will always be a specific offset from the call instruction.
- For this technique to work, when the library is linked, the GOT must be populated with all of the functions/globals defined by the other libraries at once. This works, but this would mean loading each library which could be fairly slow especially since the library may not actually call all of those functions.

Dynamic Linking

- If possible, it would be beneficial to link only the functions we need to, when they are called.
- This gives rise to a technique called “lazy binding” for when we want to call functions that are defined in other shared libraries (ie. shared1.so calls a function defined in shared2.so).
- When using “call”, instead of calling an entry of the GOT, we call an address that is an entry into a Procedure Linkage Table (PLT).
- The PLT is stored in the shared modules text segment. Each PLT entry contains a set of instructions. It is used as follows.

Dynamic Linking

- Consider the following setup for a shared1.so. fun2 has not been called yet.
- **Data Segment:**

```
GOT[0]: <addr of .dynamic>
GOT[1]: <addr of relocation entries>
GOT[2]: <addr of dynamic linker>
GOT[3]: <system startup.
GOT[4]: 0x4005c6 #eventually, this will be the entry for “fun2”
```

- **Code Segment:**

```
call 0x4005c0 # call to fun2
…
PLT[0]:  #call the dynamic linker
  4005a0: pushq GOT[1]
  4005a6: jmpq   GOT[2]
...
PLT[2]  #PLT entry for fun2
  4005c0: jmpq   GOT[4]
  4005c6: pushq  $0x1     #An identifier for fun2 like a symtab index.
  4005cb: jmpq    4005a0
```

Dynamic Linking

1. When we want to call fun2, we instead call the PLT corresponding with fun2, in this case PLT[2]
2. The first instruction jumps to the GOT entry corresponding to fun2, in this case GOT[4].
3. Since fun2 has not yet been called, GOT[4] has not yet been linked to the address of fun2. Instead, it contains an address to the instruction AFTER the jmp in PLT[2].
4. Back in PLT[2], push an id for fun2 on to the stack. Then jump to PLT[0], which is reserved for calling the dynamic linker. This calls the dynamic linker, telling it to link fun2. This will replace GOT[4] with the address of fun2.

Dynamic Linking

- **Data Segment:**
  GOT[0]: <addr of .dynamic>
  GOT[1]: <addr of relocation entries>
  GOT[2]: <addr of dynamic linker>
  GOT[3]: <system startup.
  GOT[4]: &fun2 #eventually, this will be the entry for “fun2”
- **Code Segment:**

```
call 0x4005c0 # call to fun2
…
PLT[0]:  #call the dynamic linker
  4005a0: pushq GOT[1]
  4005a6: jmpq   GOT[2]
...
PLT[2]  #PLT entry for fun2
  4005c0: jmpq   GOT[4]
  4005c6: pushq  $0x1     #An identifier for fun2 like a symtab index.
  4005cb: jmpq    4005a0
```

- The next time we call fun2...

Dynamic Linking

1. When we want to call fun2, we instead call the PLT corresponding with fun2, in this case PLT[2]
2. The first instruction jumps to the GOT entry corresponding to fun2, in this case GOT[4]
3. GOT[4] contains the address of fun2. fun2 is successfully called.

- This makes use of the fact that both the PLT and GOT locations are at fixed offsets, regardless of where the shared library is loaded.
- The data section now only consists of relative addresses and is therefore Position Independent.