238P: Operating Systems

Lecture 4: Linking and Loading
(Basic architecture of a program)

Anton Burtsev
October, 2018
What is a program?

- What parts do we need to run code?
Parts needed to run a program

- Code itself
  - By convention it's called text
- Stack
  - To call functions
- Space for variables
  - Ok... this is a bit tricky
What types of variables do you know?
What types of variables do you know?

- Global variables
  - Initialized → data section
  - Uninitalized → BSS
- Local variables
  - Stack
- Dynamic variables
  - Heap
Space for variables (3 types)

- **Global variables**
  1. `#include <stdio.h>`
  2.
  3. `char hello[] = "Hello";`
  4. `int main(int ac, char **av)`
  5. {
  6.     `static char world[] = "world!";`
  7.     `printf("%s %s\n", hello, world);`
  8.     `return 0;`
  9. }

- **Allocated in the data section**
  - It is split in initialized (non-zero), and non-initialized (zero)
  - As well as read/write, and read only data section
Space for variables (3 types)

• Local variables

1. `#include <stdio.h>
2.
3. `char hello[] = "Hello";
4. `int main(int ac, char **av)
5. {
6.     //static char world[] = "world!";
7.     `char world[] = "world!";
8.     `printf("%s %s\n", hello, world);
9.     `return 0;
10. }

• Allocated on the stack
  • Remember calling conventions?
Space for variables (3 types)

- Local variables
  1. `#include <stdio.h>`
  2. `#include <string.h>`
  3. `#include <stdlib.h>`
  4.
  5. `char hello[] = "Hello";`
  6. `int main(int ac, char **av)`
  7. {
     8.     `char world[] = "world!";`
     9.     `char *str = malloc(64);`
    10.    `memcpy(str, "beautiful", 64);`
    11.    `printf("%s %s %s\n", hello, str, world);`
    12.    `return 0;`
  13.}

- Allocated on the heap
  - Special area of memory provided by the OS from where malloc() can allocate memory
Memory layout of a process

```c
char hello = "Hello!";
main()
{
    ...
    str = malloc(64);
}
```

```c
main()
{ char world[] = "world";
    ...
}
```
Where do these areas come from?
Memory layout of a process

Compiler and linker

OS kernel

Virtual Memory

Process text

Process data

Process heap

Process stack

User-memory

Kernel-memory

2GB

0

char hello = "Hello";

main()

str = malloc(64)

main()

char world[] = "world";

...
Example program

- Compute 5 + 6

```c
#include <stdio.h>

int main(int ac, char **av)
{
    int a = 5, b = 6;
    return a + b;
}
```

- We build it like

  - I'm on 64 bit system, but want 32bit code, hence -m32

  ```bash
gcc -m32 hello-int.c
```
objdump -sd a.out

a.out: file format elf32-i386

Contents of section .text:
80483e0 d0c9e979 fffffff90 e973ffff ff5589e5 ...y.....s...U..
80483f0 83ec10c7 45f80500 0000c745 fc060000 ....E......E....
8048400 008b45fc 8b55f801 d0c9c366 90669090 ..E..U......f..f..
8048410 555731ff 5653e805 fffffff81 c3e51b00 UW1.VS..........a
8048420 0083ec1c 8b6c2430 8db30cff ffffffff61 ......1$0........a
8048430 feffffff8d 8308ffff ff29c6c1 fe0285f6 ...........)

Contents of section .rodata:
8048498 03000000 0010002000 ...

Contents of section .data:
804a014 00000000 00000000 ...

Disassembly of section .text:
...

080483ed <main>:

080483ed:           55   push   %ebp
080483ee:           89 e5 mov    %esp,%ebp
080483f0:           83 ec 10 sub    $0x10,%esp
080483f3:           c7 45 f8 05 00 00 00 movl   $0x5,-0x8(%ebp)
080483fa:           c7 45 fc 06 00 00 00 movl   $0x6,-0x4(%ebp)
08048401:           8b 45 fc mov    -0x4(%ebp),%eax
08048404:           8b 55 f8 mov    -0x8(%ebp),%edx
08048407:           01 d0 add    %edx,%eax
08048409:           c9   leave
0804840a:           c3   ret
0804840b:           66 90 xchg   %ax,%ax
0804840d:           66 90 xchg   %ax,%ax
0804840f:           90   nop

• GCC syntax, i.e.
  mov %esp, %ebp
  // EBP = ESP
Contents of section .text:
80483e0 d0c9e979 ffffff90 e973ffff ff5589e5 ...y.....s...U...
80483e0 83ec10c7 45f80500 0000c745 fe000000 ....E......E....
8048400 008b45fc 8555f801 d0c9c366 90669090 ..E..U.....f.f..
8048410 555731ff 5653e805 ffffe811 c3e51b00 UW1.VS...........
8048420 0083c1c3 8b6c2430 8db30cfe ffffe811 ......1$0.......a
8048430 feffff8d 8308ffff ff29c6c1 fe0285f6 ........)

Contents of section .rodata:
8048498 03000000 01000200

Contents of section .data:
804a014 00000000 00000000

Disassembly of section .text:
...
080483ed <main>:
   55                  push %ebp
   89 e5               mov %esp,%ebp
   83 ec 10            sub $0x10,%esp
   c7 45 f8 05 00 00 00 movl $0x5,-0x8(%ebp)
   c7 45 fc 06 00 00 00 movl $0x6,-0x4(%ebp)
   8b 45 fc            mov -0x4(%ebp),%eax
   8b 55 f8            mov -0x8(%ebp),%edx
   01 d0               add %edx,%eax
   c9                  leave
   c3                  ret
   66 90               xchg %ax,%ax
   66 90               xchg %ax,%ax
   90                  nop

GCC syntax, i.e.
mov %esp, %ebp
// EBP = ESP
Contents of section .text:
80483e0 d0c9e979 fffffff90 e973ffff ff589e5 ...y......s...U...
80483f0 83ec10c7 45f80500 0000c745 fc060000 ....E.....E....
8048400 008b45fc 8b55f801 d0c9c366 90669090 ..E..U......f.f..
8048410 555731ff 5653e805 fffffff81 c3e51b00 UW1.VS..........1$0.......a
8048420 0083ec1c 8b6c2430 8db30cff ffffe861 ......l$0.......a
8048430 feffff8d 8308ffff ff29c6c1 fe0285f6 ...........).....

Contents of section .rodata:
8048498 03000000 01000200 ........

Contents of section .data:
804a014 00000000 00000000 ........

Disassembly of section .text:
...
080483ed <main>:
80483ed:    55      push   %ebp      # Maintain the stack frame
80483ee:    89 e5    mov    %esp,%ebp
80483f0:    83 ec 10 sub    $0x10,%esp
80483f3:    c7 45 f8 05 00 00 00 movl   $0x5,-0x8(%ebp)
80483fa:    c7 45 fc 06 00 00 00 movl   $0x6,-0x4(%ebp)
8048401:    8b 45 fc mov    -0x4(%ebp),%eax
8048404:    8b 55 f8 mov    -0x8(%ebp),%edx
8048407:    01 d0 add    %edx,%eax
8048409:    c9      leave
804840a:    c3      ret
804840b:    66 90  xchg   %ax,%ax
804840d:    66 90  xchg   %ax,%ax
804840f:    90      nop

• GCC syntax, i.e. mov %esp, %ebp
  // EBP = ESP
Contents of section .text:

```
80483ed <main>:
  80483ed:        55                      push   %ebp
  80483ee:        89 e5                   mov    %esp,%ebp
  80483f0:        83 ec 10                 sub    $0x10,%esp     # Allocate space for a and b
  80483f3:        c7 45 f8 05 00 00 00    movl   $0x5,-0x8(%ebp)
  80483fa:        c7 45 fc 06 00 00 00    movl   $0x6,-0x4(%ebp)
  8048401:        8b 45 fc                mov    -0x4(%ebp),%eax
  8048404:        8b 55 f8                mov    -0x8(%ebp),%edx
  8048407:        01 d0                   add    %edx,%eax
  8048409:        c9                      leave
  804840a:        c3                      ret
  804840b:        66 90                   xchg   %ax,%ax
  804840d:        66 90                   xchg   %ax,%ax
  804840f:        90                      nop
```

GCC syntax, i.e.

```
mov %esp, %ebp
// EBP = ESP
```
Contents of section .text:

80483e0 d0c9e979 ffffff90 e973ffff ff5589e5 ...y.....s...U..
80483f0 83ec10c7 45f80500 0000c745 fc060000 ....E......E...
8048400 008b45fc 8b55f801 d0c9c366 90669090 ..E..U......f.f..
8048410 555731ff 5653e805 ffffffff81 c3e51b00 UW1.VS.........
8048420 0083ec1c 8b6c2430 8db30cff ffff8e61 ......1$0........a
8048430 feffff8d 8308ffff ff29c6c1 fe0285f6 ........)...)

Contents of section .rodata:

8048498 03000000 01000200 ........

Contents of section .data:

804a014 00000000 00000000 ........

Disassembly of section .text:

080483ed <main>:

```
  push %ebp
  mov %esp,%ebp
  sub $0x10,%esp     # Allocate space for a and b
  movl $0x5,-0x8(%ebp)
  movl $0x6,-0x4(%ebp)
  movl %esp,%eax
  movl -0x8(%ebp),%edx
  add %edx,%eax
  leave
  ret
```

GCC syntax, i.e.

```
mov %esp, %ebp
// EBP = ESP
```
Contents of section .text:
80483e0 d0c9e979 fffffff90 e973ffff ff5589e5 ...y.....s...U..
80483f0 83ec10c7 45f80500 0000c745 fc060000 .....E.......E....
8048400 008b45fc 8b55f801 d0c9c366 90669090 ..E..U......f.f..
8048410 555731ff 5653e805 fffffff81 c3e51b00 UW1.VS..........U
8048420 0083ec1c 8b6c2430 8db30cff ffffe861 .....l$0........
8048430 feffff8d 8308ffff ff29c6c1 fe0285f6 ..........)......

Contents of section .rodata:
8048498 03000000 01000200 ........

Contents of section .data:
804a014 00000000 00000000 ........

Disassembly of section .text:
...
080483ed <main>:
80483ed: 55 push %ebp
80483ee: 89 e5 mov %esp,%ebp
80483f0: 83 ec 10 sub $0x10,%esp
80483f3: c7 45 f8 05 00 00 00 movl $0x5,-0x8(%ebp) # Initialize a = 5
80483fa: c7 45 fc 06 00 00 00 movl $0x6,-0x4(%ebp) # Initialize b = 6
8048401: 8b 45 fc mov -0x4(%ebp),%eax
8048404: 8b 55 f8 mov -0x8(%ebp),%edx
8048407: 01 d0 add %edx,%eax
8048409: c9 leave
804840a: c3 ret
804840b: 66 90 xchg %ax,%ax
804840d: 66 90 xchg %ax,%ax
804840f: 90 nop

• GCC syntax, i.e.
mov %esp, %ebp
// EBP = ESP
a.out: file format elf32-i386

Contents of section .text:
80483e0 d0c9e979 ffffff90 e973ffff ff5589e5 ...y......s...U..  
80483f0 83ec10c7 45f80500 0000c745 fc060000 .....E......E....  
8048400 008b45fc 8b55f801 d0c9c366 90669090 ..E..U......f.f..  
8048410 555731ff 5653e805 ffffff81 c3e51b00 UW1.VS..........  
8048420 0083ec1c 8b6c2430 8db30cff fffffe861 ......1$0........a  
8048430 feffff8d 8308ffff ff29c6c1 fe0285f6 ........)...……

Contents of section .rodata:
8048498 03000000 01000200                    ........

Contents of section .data:
804a014 00000000 00000000                    ........

Disassembly of section .text:
...

080483ed <main>:
080483ed:  55    push %ebp
080483ee:  89 e5 mov %esp,%ebp
080483f0:  83 ec 10 sub $0x10,%esp
080483f3:  c7 45 fc 05 00 00 movl $0x5,-0x8(%ebp) # Initialize a = 5
080483f8:  c7 45 fc 06 00 00 movl $0x6,-0x4(%ebp) # Initialize b = 6
08048401:  8b 45 fc mov -0x4(%ebp),%eax
08048404:  8b 55 f8 mov -0x8(%ebp),%edx
08048407:  01 d0 add %edx,%eax
08048409:  c9 leave
0804840a:  c3 ret
0804840b:  66 90 xchg %ax,%ax
0804840d:  66 90 xchg %ax,%ax
0804840f:  90 nop

• GCC syntax, i.e.
  mov %esp, %ebp
  // EBP = ESP
objdump -sd a.out

Contents of section .text:
80483e0 d0c9e979 ffffff90 e973ffff ff589e5 ...y.....s...U..
80483f0 83ec10c7 45f80500 0000c745 fc060000 ....E....E....
8048400 008b45fc 8b55f801 d0c9c366 90669090 ..E..U.....f..f..
8048410 555731ff 5653e805 ffffff81 c3e51b00 UW1.VS............
8048420 0083ec1c 8b6c2430 8db30cff ffffe861 ......1$0........a
8048430 feffff8d 8308ffff ff29c6c1 fe0285f6 ........)...)

Contents of section .rodata:
8048498 03000000 01000200                    ........

Contents of section .data:
804a014 00000000 00000000                    ........

Disassembly of section .text:
...
080483ed <main>:
  80483ed:       55                      push   %ebp
  80483ee:       89 e5                   mov    %esp,%ebp
  80483f0:       83 ec                   sub    $0x10,%esp
  80483f3:       c7 45 f8 f05 00 00 00 00 movl   $0x5,-0x8(%ebp) # Initialize a = 5
  80483fa:       c7 45 fc                movl   $0x6,-0x4(%ebp) # Initialize b = 6
  8048401:       8b 45 fc                mov    -0x4(%ebp),%eax
  8048404:       8b 55 f8                mov    -0x8(%ebp),%edx
  8048407:       01 d0                   add    %edx,%eax
  8048409:       c9                      leave
  804840a:       c3                      ret
  804840b:       66 90                   xchg   %ax,%ax
  804840d:       66 90                   xchg   %ax,%ax
  804840f:       90                      nop

- GCC syntax, i.e.
  mov %esp, %ebp
  // EBP = ESP
Contents of section .text:

```
80483e0 d0c9e979 fffffff90 e973ffff ff5589e5 y...y...s...U...
80483f0 83ec10c7 45f80500 0000c745 fc060000 E.....E....
8048400 08b45fc 8b55f801 d0c9c366 90669090 E..U.....f.f..
8048410 555731ff 5653e805 fffffff81 c3e51b00 UW1.VS...........
8048420 008ec1c 8b6c2430 8db30cff ffffe861 1$0.......a
8048430 feffff8d 8308ffff ff29c6c1 fe0285f6 .........).....
```

Contents of section .rodata:

```
8048498 03000000 01000200 ........
```

Contents of section .data:

```
804a014 00000000 00000000 ........
```

Disassembly of section .text:

```
080483ed <main>:
  80483ed:       55                      push   %ebp
  80483ee:       89 e5                   mov    %esp,%ebp
  80483f0:       83 ec 10                sub    $0x10,%esp
  80483f3:       c7 45 f8 05 00 00 00    movl   $0x5,-0x8(%ebp)
  80483fa:       c7 45 fc 06 00 00 00    movl   $0x6,-0x4(%ebp)
  8048401:       8b 45 fc 06 00 00 00    movl   -0x4(%ebp),%eax # Move b into %eax
  8048404:       8b 55 f8 05 00 00 00    movl   -0x8(%ebp),%edx # Move a into %edx
  8048407:       01 d0                   add    %edx,%eax
  8048409:       c9                      leave
  804840a:       ret                      ret
  804840b:       66 90                   xchg   %ax,%ax
  804840d:       66 90                   xchg   %ax,%ax
  804840f:       90                      nop
```

- GCC syntax, i.e.

```
mov %esp, %ebp
// EBP = ESP
```
Contents of section .text:

```
080483ed <main>:
  80483ed:      55                      push   %ebp
  80483ee:      89 e5                   mov    %esp,%ebp
  80483f0:      83 ec 10                sub    $0x10,%esp
  80483f3:      c7 45 f8 05 00 00 00    movl   $0x5,-0x8(%ebp)
  80483fa:      c7 45 fc 06 00 00 00    movl   $0x6,-0x4(%ebp)
  8048401:      8b 45 fc                mov    -0x4(%ebp),%eax
  8048404:      8b 55 f8                mov    -0x8(%ebp),%edx
  8048407:      01 d0                    add    %edx,%eax      # a + b
  8048409:      c9                      leave
  804840a:      c3                      ret
  804840b:      66 90                   xchg   %ax,%ax
  804840d:      66 90                   xchg   %ax,%ax
  804840f:      90                      nop
```

Contents of section .rodata:

```
08048498 03000000 01000200                    ........
```

Contents of section .data:

```
0804a014 00000000 00000000                    ........
```

Disassembly of section .text:

```
...
Contents of section .text:
80483e0 d0c9e979 ffffff90 e973ffff ff5589e5 ...y......s....U...
80483f0 83ec10c7 45f80500 0000c745 fc060000 ....E......E....
8048400 008b45fc 8b55f801 d0c9c366 90669090 ..E..U.....f.f..
8048410 555731ff 5653e805 ffffff81 c3e51b00 UW1.VS.......a
8048420 0083ec1c 8b6c2430 8db30cff fffffe81 ......1$0.......a
8048430 feffff8d 8308ffff ff29c6c1 fe0285f6 ........)...)

Contents of section .rodata:
8048498 03000000 01000200 ........

Contents of section .data:
804a014 00000000 00000000 ........

Disassembly of section .text:
...
080483ed <main>:
  80483ed:      55                      push   %ebp
  80483ee:      89 e5                   mov    %esp,%ebp
  80483f0:      83 ec 10                sub    $0x10,%esp
  80483f3:      c7 45 f8 05 00 00 00    movl   $0x5,-0x8(%ebp)
  80483fa:      c7 45 fc 06 00 00 00    movl   $0x6,-0x4(%ebp)
  8048401:      8b 45 fc                mov    -0x4(%ebp),%eax
  8048404:      8b 55 f8                mov    -0x8(%ebp),%edx
  8048407:      01 d0                   add    %edx,%eax
  8048409:      c9                      leave
  804840a:      c3                      ret
  804840b:      66 90                   xchg   %ax,%ax
  804840d:      66 90                   xchg   %ax,%ax
  804840f:      90                      nop

GCC syntax, i.e.
mov %esp, %ebp
// EBP = ESP

# Pop the frame ESP = EBP
# return
a.out:     file format elf32-i386

Contents of section .text:
80483e0 d0c9e979 ffffff90 e973ffff ff5589e5 ...y......s...U..
80483f0 83ec10c7 45f80500 0000c745 fc060000 ....E......E....
8048400 008b45fc 8b55f801 d0c9c366 90669090 ..E..U......f.f..
8048410 555731ff 5653e805 ffffff81 c3e51b00 UW1.VS.........
8048420 0083ec1c 8b6c2430 8db30cff ffffe861 .....1$0.......a
8048430 feffff8d 8308ffff ff29c6c1 fe0285f6 ........)......

Contents of section .rodata:
8048498 03000000 01000200                    ........

Contents of section .data:
804a014 00000000 00000000                    ........

Disassembly of section .text:
...
080483ed <main>:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>080483ed</td>
<td>push %ebp</td>
</tr>
<tr>
<td>080483ee</td>
<td>mov %esp,%ebp</td>
</tr>
<tr>
<td>080483f0</td>
<td>sub $0x10,%esp</td>
</tr>
<tr>
<td>080483f3</td>
<td>movl $0x5,-0x8(%ebp)</td>
</tr>
<tr>
<td>080483fa</td>
<td>movl $0x6,-0x4(%ebp)</td>
</tr>
<tr>
<td>08048401</td>
<td>mov -0x4(%ebp),%eax</td>
</tr>
<tr>
<td>08048404</td>
<td>mov -0x8(%ebp),%edx</td>
</tr>
<tr>
<td>08048407</td>
<td>add %edx,%eax</td>
</tr>
<tr>
<td>08048409</td>
<td>leave</td>
</tr>
<tr>
<td>0804840a</td>
<td>ret</td>
</tr>
<tr>
<td>0804840b</td>
<td>xchg %ax,%ax</td>
</tr>
<tr>
<td>0804840d</td>
<td>xchg %ax,%ax</td>
</tr>
<tr>
<td>0804840f</td>
<td>nop</td>
</tr>
</tbody>
</table>

• GCC syntax, i.e.
  mov %esp, %ebp
// EBP = ESP
11.5 Alignment of code

**Most microprocessors fetch code in aligned 16-byte or 32-byte blocks.** If an important subroutine entry or jump label happens to be near the end of a 16-byte block then the microprocessor will only get a few useful bytes of code when fetching that block of code. It may have to fetch the next 16 bytes too before it can decode the first instructions after the label. This can be avoided by aligning important subroutine entries and loop entries by 16.

... Aligning a subroutine entry is as simple as putting as many NOP 's as needed before the subroutine entry to make the address divisible by 8, 16, 32 or 64, as desired.
Load program in memory

char hello = "Hello";
main()
{
    ...
    str = malloc(64);
    ...
}

main()
{
    char *world = "world";
    ...
}

Allocate pages for stack and heap
read program code and data

Kernel-memory
We however build programs from multiple files

Part of the xv6 Makefile

```
bootblock: bootasm.S bootmain.c

$(CC) $(CFLAGS) -fno-pic -O -nostdinc -I. -c bootmain.c
$(CC) $(CFLAGS) -fno-pic -nostdinc -I. -c bootasm.S
$(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 -o bootblock.o bootasm.o bootmain.o
$(OBJDUMP) -S bootblock.o > bootblock.asm
$(OBJCOPY) -S -O binary -j .text bootblock.o bootblock
./sign.pl bootblock
```
Linking and loading

- Linking
  - Combining multiple code modules into a single executable
  - E.g., use standard libraries in your own code

- Loading
  - Process of getting an executable running on the machine
• Input: object files (code modules)
• Each object file contains
  • A set of segments
    – Code
    – Data
  • A symbol table
    – Imported & exported symbols
• Output: executable file, library, etc.
Object A calls B, C, and D

Object B calls C and E

Linker

Library 1
- C
- D
- X
- Y

Library 2
- E
- F

Executable file
Why linking?
Why linking?

- **Modularity**
  - Program can be written as a collection of modules
  - Can build libraries of common functions

- **Efficiency**
  - Code compilation
    - Change one source file, recompile it, and re-link the executable
  - Space efficiency
    - Share common code across executables
    - On disk and in memory
Two path process

Path 1: scan input files
- Identify boundaries of each segment
- Collect all defined and undefined symbol information
- Determine sizes and locations of each segment

Path 2
- Adjust memory addresses in code and data to reflect relocated segment addresses
• Save a into b, e.g., \( b = a \)

\[
\text{mov} \ a, \ %\text{eax} \\
\text{mov} \ %\text{eax}, \ b
\]

• Generated code
  
  • a is defined in the same file at 0x1234, \textit{b is imported}
  
  • Each instruction is 1 byte opcode + 4 bytes address

\[
\text{A1 34 12 00 00 mov a, %eax} \\
\text{A3 00 00 00 00 00 mov %eax, b}
\]
• Save a into b, e.g., \( b = a \)

\[
\begin{align*}
\text{mov } a, \%eax \\
\text{mov} \%eax, b
\end{align*}
\]

Example

Generated code

- a is defined in the same file at 0x1234, \textbf{b is imported}
- Each instruction is 1 byte opcode + 4 bytes address

\[
\begin{align*}
\text{A1} & \ 34 \ 12 \ 00 \ 00 \ \text{mov } a, \%eax \\
\text{A3} & \ 00 \ 00 \ 00 \ 00 \ \text{mov } \%eax, b
\end{align*}
\]
• Save a into b, e.g., \( b = a \)

\[
\text{mov } a, \%eax
\]

• \( a \) is defined in the same file at \( 0x1234 \), \( b \) \textbf{is imported}

• Each instruction is 1 byte opcode + 4 bytes address

\[
\begin{align*}
A1 & \quad 34 \ 12 \ 00 \ 00 \quad \text{mov } a, \%eax \\
A3 & \quad 00 \ 00 \ 00 \ 00 \ 00 \quad \text{mov } \%eax, b
\end{align*}
\]
• Save a into b, e.g., b = a
  
  mov a, %eax
  mov %eax, b

• Generated code
  
  • a is defined in the same file at 0x1234, b is imported
  • Each instruction is 1 byte opcode + 4 bytes address

  A1 34 12 00 00 mov a, %eax
  A3 00 00 00 00 mov %eax, b

  • b is imported, we don't know yet where it will be
• Save a into b, e.g., \( b = a \)

\[
\begin{align*}
\text{mov a, } &\%eax \\
\text{mov } &\%eax, b
\end{align*}
\]

• Generated code

  • a is defined in the same file at 0x1234, **b is imported**
  • Each instruction is 1 byte opcode + 4 bytes address

\[
\begin{align*}
A1 \ 34 \ 12 \ 00 \ 00 \ \text{mov a, } &\%eax \\
A3 \ 00 \ 00 \ 00 \ 00 \ \text{mov } &\%eax, b
\end{align*}
\]

• Assume that a is relocated by \( 0x10000 \) bytes, and b is found at 0x9a12

\[
\begin{align*}
A1 \ 34 \ 12 \ 01 \ 00 \ \text{mov a, } &\%eax \\
A3 \ 12 \ 9A \ 00 \ 00 \ \text{mov } &\%eax, b
\end{align*}
\]
- Save a into b, e.g., \( b = a \)
  
  \begin{verbatim}
  mov a, %eax  
  mov %eax, b 
  \end{verbatim}

- Generated code
  
  - a is defined in the same file at 0x1234, **b is imported**
  - Each instruction is 1 byte opcode + 4 bytes address

  A1  34 12 00 00  mov a, %eax  
  A3  00 00 00 00  mov %eax, b  

  - Assume that a is relocated by 0x10000 bytes, and b is found at **0x9a12**

  A1  34 12 01 00  mov a, %eax  
  A3  12 9A 00 00  mov %eax, b  

• Source file m.c

```c
1    extern void a(char *);
2    int main(int ac, char **av)
3    {
4        static char string[] = "Hello, world!\n";
5        a(string);
6    }
```

• Source file a.c

```c
1    #include <unistd.h>
2    #include <string.h>
3    void a(char *s)
4    {
5        write(1, s, strlen(s));
6    }
```
More realistic example

• Source file m.c

```c
extern void a(char *);
int main(int ac, char **av) {
    static char string[] = "Hello, world!\n";
    a(string);
}
```

• Source file a.c

```c
#include <unistd.h>
#include <string.h>
void a(char *s) {
    write(1, s, strlen(s));
}
```
More realistic example

● Source file m.c

```c
1 extern void a(char *);
2 int main(int ac, char **av)
3 {
4     static char string[] = "Hello, world!\n";
5     a(string);
6 }
```

● Source file a.c

```c
1 #include <unistd.h>
2 #include <string.h>
3 void a(char *s)
4 {
5     write(1, s, strlen(s));
6 }
```
More realistic example

Sections:

<table>
<thead>
<tr>
<th>Idx</th>
<th>Name</th>
<th>Size</th>
<th>VMA</th>
<th>LMA</th>
<th>File off</th>
<th>Algn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>.text</td>
<td>00000010</td>
<td>00000000</td>
<td>00000000</td>
<td>00000020</td>
<td>2**3</td>
</tr>
<tr>
<td>1</td>
<td>.data</td>
<td>00000010</td>
<td>00000010</td>
<td>00000010</td>
<td>00000030</td>
<td>2**3</td>
</tr>
</tbody>
</table>

Disassembly of section .text:

00000000 <_main>:

0: 55          pushl %ebp
1: 89 e5       movl %esp,%ebp
3: 68 10 00 00 00 pushl $0x10
4: 32 .data
8: e8 f3 ff ff ff call 0
9: DISP32 _a
d: c9          leave
e: c3          ret
...
More realistic example

- Two sections:
  - Text (0x10 – 16 bytes)
  - Data (16 bytes)

Sections:

<table>
<thead>
<tr>
<th>Idx</th>
<th>Name</th>
<th>Size</th>
<th>VMA</th>
<th>LMA</th>
<th>File off</th>
<th>Algn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>.text</td>
<td>00000010</td>
<td>00000000</td>
<td>00000000</td>
<td>00000020</td>
<td>2**3</td>
</tr>
<tr>
<td>1</td>
<td>.data</td>
<td>00000010</td>
<td>00000010</td>
<td>00000010</td>
<td>00000030</td>
<td>2**3</td>
</tr>
</tbody>
</table>

Disassembly of section .text:

00000000 <_main>:

0: 55  pushl %ebp
1: 89 e5  movl %esp,%ebp
3: 68 10 00 00 00  pushl $0x10
4: 32  .data
8: e8 f3 ff ff ff  call 0
9: DISP32 _a
d: c9  leave
e: c3  ret
...

More realistic example

Two sections:
- Text starts at 0x0
- Data starts at 0x10

Sections:

<table>
<thead>
<tr>
<th>Idx</th>
<th>Name</th>
<th>Size</th>
<th>VMA</th>
<th>LMA</th>
<th>File off</th>
<th>Algn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>.text</td>
<td>00000010</td>
<td>00000000</td>
<td>00000000</td>
<td>00000020</td>
<td>2**3</td>
</tr>
<tr>
<td>1</td>
<td>.data</td>
<td>00000010</td>
<td>00000010</td>
<td>00000010</td>
<td>00000030</td>
<td>2**3</td>
</tr>
</tbody>
</table>

Disassembly of section .text:

00000000 <_main>:

0: 55         pushl %ebp
1: 89 e5      movl %esp,%ebp
3: 68 10 00 00 00 pushl $0x10
4: 32 .data
8: e8 f3 ff ff ff ff call 0
   9: DISP32 _a
d: c9         leave
   e: c3       ret
...
More realistic example

Sections:

<table>
<thead>
<tr>
<th>Idx</th>
<th>Name</th>
<th>Size</th>
<th>VMA</th>
<th>LMA</th>
<th>File off</th>
<th>Algn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>.text</td>
<td>00000010</td>
<td>00000000</td>
<td>00000000</td>
<td>00000020</td>
<td>2**3</td>
</tr>
<tr>
<td>1</td>
<td>.data</td>
<td>00000010</td>
<td>00000010</td>
<td>00000010</td>
<td>00000030</td>
<td>2**3</td>
</tr>
</tbody>
</table>

Disassembly of section .text:

00000000 <_main>:

0: 55           pushl %ebp
1: 89 e5        movl %esp,%ebp
3: 68 10 00 00 00 pushl $0x10

4: 32 .data

8: e8 f3 ff ff ff ff call 0

9: DISP32 _a
d: c9           leave
e: c3           ret

...
More realistic example

Sections:

<table>
<thead>
<tr>
<th>Idx</th>
<th>Name</th>
<th>Size</th>
<th>VMA</th>
<th>LMA</th>
<th>File off</th>
<th>Algn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>.text</td>
<td>00000010</td>
<td>00000000</td>
<td>00000000</td>
<td>00000020</td>
<td>2**3</td>
</tr>
<tr>
<td>1</td>
<td>.data</td>
<td>00000010</td>
<td>00000010</td>
<td>00000010</td>
<td>00000030</td>
<td>2**3</td>
</tr>
</tbody>
</table>

Disassembly of section .text:

00000000 <_main>:

0: 55             pushl %ebp
1: 89 e5          movl %esp,%ebp
3: 68 10 00 00 00 00 pushl $0x10 # push string on the stack
4: 32 .data
8: e8 f3 ff ff ff 00 call 0
9: DISP32 _a
d: c9             leave
e: c3             ret
...

- First relocation entry
- Marks pushl 0x10
- 0x10 is beginning of the data section
- and address of the string
More realistic example

- Source file m.c

```c
1   extern void a(char *);
2   int main(int ac, char **av)
3   {
4       static char string[] = "Hello, world!\n";
5       a(string);
6   }
```

- Source file a.c

```c
1   #include <unistd.h>
2   #include <string.h>
3   void a(char *s)
4   {
5       write(1, s, strlen(s));
6   }
```
More realistic example

Sections:

<table>
<thead>
<tr>
<th>Idx</th>
<th>Name</th>
<th>Size</th>
<th>VMA</th>
<th>LMA</th>
<th>File off</th>
<th>Algn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>.text</td>
<td>00000010</td>
<td>00000000</td>
<td>00000000</td>
<td>00000020</td>
<td>2**3</td>
</tr>
<tr>
<td>1</td>
<td>.data</td>
<td>00000010</td>
<td>00000010</td>
<td>00000010</td>
<td>00000030</td>
<td>2**3</td>
</tr>
</tbody>
</table>

Disassembly of section .text:

```
00000000 <_main>:
   0: 55           pushl %ebp
   1: 89 e5        movl %esp,%ebp
   3: 68 10 00 00 00 pushl $0x10
   4: 32 .data     
   8: e8 f3 ff ff ff ff call 0
   9: DISP32 _a    
  d: c9           leave
  e: c3           ret
...
```

- Second relocation entry
  - Marks call
  - 0x0 – address is unknown
More realistic example

- Two sections:
  - Text (0 bytes)
  - Data (28 bytes)
More realistic example

Sections:

<table>
<thead>
<tr>
<th>Idx</th>
<th>Name</th>
<th>Size</th>
<th>VMA</th>
<th>LMA</th>
<th>File off</th>
<th>Algn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>.text</td>
<td>0000001c</td>
<td>00000000</td>
<td>00000000</td>
<td>000000020</td>
<td>2**2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>CONTENTS, ALLOC, LOAD, RELOC, CODE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>.data</td>
<td>00000000</td>
<td>0000001c</td>
<td>0000001c</td>
<td>00000003c</td>
<td>2**2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>CONTENTS, ALLOC, LOAD, DATA</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Disassembly of section .text:

00000000 <_a>:

0: 55               pushl %ebp
1: 89 e5            movl %esp,%ebp
3: 53               pushl %ebx
4: 8b 5d 08         movl 0x8(%ebp),%ebx
7: 53               pushl %ebx
8: e8 f3 ff ff ff   call 0
\hspace{0.5cm} \text{\underline{9: DISP32 \_strlen}}

d: 50               pushl %eax
e: 53               pushl %ebx
f: 6a 01            pushl $0x1
11: e8 ea ff ff ff  call 0
\hspace{0.5cm} \text{\underline{12: DISP32 \_write}}

16: 8d 65 fc        leal -4(%ebp),%esp
19: 5b              popl %ebx
1a: c9              leave
1b: c3              ret

- Two relocation entries:
  - strlen()
  - write()
Producing an executable

- Combine corresponding segments from each object file
  - Combined text segment
  - Combined data segment
- Pad each segment to 4KB to match the page size
Multiple object files

Inputs

0
600

Module A

0
400

Module B

0
500

Module C

Outputs

Code from A

Code from B

Code from C
Sections:
Idx Name  Size     VMA      LMA      File off Algn
 0 .text 00000fe0 00001020 00001020 00000020 2**3
 1 .data 00001000 00002000 00002000 00001000 2**3
 2 .bss  00000000 00003000 00003000 00000000 2**3

Disassembly of section .text:
00001020 <start-c>:
  ...
  1092: e8 0d 00 00 00 call 10a4 <_main>
  ...
000010a4 <_main>:
  10a7: 68 24 20 00 00 pushl $0x2024
  10ac: e8 03 00 00 00 call 10b4 <_a>
  ...
000010b4 <_a>:
  10bc: e8 37 00 00 00 call 10f8 <_strlen>
  ...
  10c3: 6a 01 pushl $0x1
  10c5: e8 a2 00 00 00 call 116c <_write>
  ...
000010f8 <_strlen>:
  ...
0000116c <_write>:
Sections:

<table>
<thead>
<tr>
<th>Idx</th>
<th>Name</th>
<th>Size</th>
<th>VMA</th>
<th>LMA</th>
<th>File off</th>
<th>Algn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>.text</td>
<td>0</td>
<td>0x000fe0</td>
<td>0x0001020</td>
<td>0x0001020</td>
<td>0x0000020</td>
</tr>
<tr>
<td>1</td>
<td>.data</td>
<td>0</td>
<td>0x0001000</td>
<td>0x0002000</td>
<td>0x0002000</td>
<td>0x0001000</td>
</tr>
<tr>
<td>2</td>
<td>.bss</td>
<td>0</td>
<td>0x0000000</td>
<td>0x0003000</td>
<td>0x0003000</td>
<td>0x0000000</td>
</tr>
</tbody>
</table>

Disassembly of section .text:

00001020 <start-c>:
...
1092: e8 0d 00 00 00 call 10a4 <_main>
...
000010a4 <_main>:
10a7: 68 24 20 00 00 pushl $0x2024
10ac: e8 03 00 00 00 call 10b4 <_a>
...
000010b4 <_a>:
10bc: e8 37 00 00 00 call 10f8 <_strlen>
...
10c3: 6a 01 pushl $0x1
10c5: e8 a2 00 00 00 call 116c <_write>
...
000010f8 <_strlen>:
...
0000116c <_write>:
...

- Relative to EIP address
- Hence 3

Linked executable
Tasks involved

- **Program loading**
  - Copy a program from disk to memory so it is ready to run
    - Allocation of memory
    - Setting protection bits (e.g. read only)
- **Relocation**
  - Assign load address to each object file
  - Adjust the code
- **Symbol resolution**
  - Resolve symbols imported from other object files
Object files
Object files

- Conceptually: five kinds of information
  - Header: code size, name of the source file, creation date
  - Object code: binary instruction and data generated by the compiler
  - Relocation information: list of places in the object code that need to be patched
  - Symbols: global symbols defined by this module
    - Symbols to be imported from other modules
  - Debugging information: source file and file number information, local symbols, data structure description
Example: UNIX A.OUT

- Small header
- Text section
  - Executable code
- Data section
  - Initial values for static data
• A.OUT header

```c
int a_magic;    // magic number
int a_text;    // text segment size
int a_data;    // initialized data size
int a_bss;     // uninitialized data size
int a_syms;    // symbol table size
int a_entry;   // entry point
int a_trsize;  // text relocation size
int a_drsize;  // data relocation size
```
A.OUT loading
A.OUT loading

• Read the header to get segment sizes
• Check if there is a shareable code segment for this file
  – If not, create one,
  – Map into the address space,
  – Read segment from a file into the address space
• Create a private data segment
  – Large enough for data and BSS
  – Read data segment, zero out the BSS segment
• Create and map stack segment
  – Place arguments from the command line on the stack
• Jump to the entry point
Types of object files

- Relocatable object files (.o)
- Static libraries (.a)
- Shared libraries (.so)
- Executable files

- We looked at A.OUT, but Unix has a general format capable to hold all of these files
ELF

Elf header
- Magic number, type (.o, exec, .so), machine, byte ordering, etc.

Segment header table
- Page size, virtual addresses memory segments (sections), segment sizes.

.text section
- Code

.data section
- Initialized global variables

.bss section
- Uninitialized global variables
- "Block Started by Symbol"
- "Better Save Space"
- Has section header but occupies no space
ELF (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
Initializers and finalizers

- C++ needs a segment for invoking constructors for static variables
  - List of pointers to startup routines
    - Startup code in every module is put into an anonymous startup routine
    - Put into a segment called .init

- Problem
  - Order matters
  - Ideally you should track dependencies
    - This is not done
  - Simple hack
    - System libraries go first (.init), then user (.ctor)
Static libraries
Libraries

- Conceptually a library is
  - Collection of object files

- UNIX uses an archive format
  - Remember the `ar` tool
  - Can support collections of any objects
  - Rarely used for anything instead of libraries
Creating a static library

- `atoui.c` compiled by `gcc -c` to `atoui.o`
- `printf.c` compiled by `gcc -c` to `printf.o`
- `random.c` compiled by `gcc -c` to `random.o`

Combined using `ar` to create the static library `libc.a`
Searching libraries

• First linker path needs resolve symbol names into function locations

• To improve the search library formats add a directory
  • Map names to member positions
Shared libraries (.so or .dll)
Motivation

• 1000 programs in a typical UNIX system
• 1000 copies of printf

• How big is printf() actually?
Motivation

• Disk space
  • 2504 programs in /usr/bin on my Linux laptop
    - `ls /usr/bin | wc -l`
  • `printf()` is a large function
  • Handles conversion of multiple types to strings
    - 5-10K
  • This means 10-25MB of disk can be wasted just on `printf()`

• Runtime memory costs are
  • 5-10K times the number of running programs
  • 250 programs running on my Linux laptop
    - `ps -aux | wc -l`
    - 1MB-2.5MB – huge number for most systems 15-20 years ago
Shared libraries

• Motivation
  • Share code of a library across all processes
    – E.g. libc is linked by all processes in the system
  • Code section should remain identical
    – To be shared read-only
  • What if library is loaded at different addresses?
    – Remember it needs to be relocated
Example: size of a statically vs dynamically linked program

- On Ubuntu 16.04 (gcc 5.4.0, libc 2.23)
  - Statically linked trivial example
    - `gcc -m32 -static hello-int.c -o test`
    - 725KB
  - Dynamically linked trivial example
    - `gcc -m32 hello-int.c -o test`
    - 7KB
Position independent code

(Parts adapted from Eli Bendersky)

Position independent code

• Motivation
  • Share code of a library across all processes
    – E.g. libc is linked by all processes in the system
  • Code section should remain identical
    – To be shared read-only
  • What if library is loaded at different addresses?
    – Remember it needs to be relocated
Position independent code (PIC)

- Main idea:
  - Generate code in such a way that it can work no matter where it is located in the address space
  - Share code across all address spaces
What needs to be changed?

- Can stay untouched
  - Local jumps and calls are relative
  - Stack data is relative to the stack
- Needs to be modified
  - Global variables
  - Imported functions
Example

000010a4 <_main>:
  10a4: 55      pushl %ebp
  10a5: 89 e5   movl %esp,%ebp
  10a7: 68 10 00 00 00 pushl $0x10
    10a8: 32 .data
  10ac: e8 03 00 00 00 call 10b4 <_a>
   ...
000010b4 <_a>:
  10bc: e8 37 00 00 00 call 10f8 <_strlen>
    ...
  10c3: 6a 01 pushl $0x1
  10c5: e8 a2 00 00 00 call 116c <_write>
   ...

- Reference to a data section
- Code and data sections can be moved around
Example

000010a4 <_main>:
  10a4: 55       pushl %ebp
  10a5: 89 e5    movl %esp,%ebp
  10a7: 68 10 00 00 00 pushl $0x10
       10a8: 32 .data
  10ac: e8 03 00 00 00 call 10b4 <_a>

...  
000010b4 <_a>:
  10bc: e8 37 00 00 00 call 10f8 <_strlen>

...  
  10c3: 6a 01 pushl $0x1
  10c5: e8 a2 00 00 00 call 116c <_write>

...  

- Local function invocations use relative addresses
- No need to relocate
Position independent code

- How would you build it?
Position independent code

• How would you build it?
• Main idea:
  • Add additional layer of indirection to all
    – Global data
    – Function
    – ...references in the code
Position independent code

• Main insight
  • Code sections are followed by data sections
  • The distance between code and data remains constant even if code is relocated
    - Linker knows the distance
    - Even if it combines multiple code sections together
Insight 1: Constant offset between text and data sections
Global offset table (GOT)

• Insight #2:
  • Instead of referring to a variable by its absolute address
    - Which would require a relocation
  • Refer through GOT
Global offset table (GOT)

- **GOT**
  - Table of addresses
  - Each entry contains absolute address of a variable
  - GOT is patched by the linker at relocation time
How to find position of the code in memory at run time?
How to find position of the code in memory at run time?

- Is there an x86 instruction that does this?
  - i.e., give me my current code address

- x86 32bit architecture requires absolute addresses for `mov` instructions
  - No relative addresses allowed

- There is no instruction to learn the value of EIP
  - Instruction pointer
How to find position of the code in memory at run time?

• Simple trick

```assembly
    call L2
L2: popl %ebx
```

• Call next instruction
  • Saves EIP on the stack
  • EIP holds current position of the code
  • Use popl to fetch EIP into a register
Load address unknown at link time

Code segment

XX0000

XX0010

GOT

XX1000

Data segment

Fixed distance from code to GOT

call L2
L2: pop %bx
    add $FF0,%bx
What did we gain?

- Processes can share code
- Each have private GOT
- Why is it better?
  - GOT is in the data section, private to each process anyway
    - We saved memory
  - We saved some linking time too
    - GOT is patched per variable, not per variable reference
    - There are many references to the same variable in the code
    - It takes some time to relocate
    - We saved this time
int myglob = 42;

int ml_func(int a, int b)
{
    return myglob + a + b;
}

0000043c <ml_func>:
  43c:  55                      push   ebp
  43d:  89 e5                   mov    ebp,esp
  43f:  e8 16 00 00 00          call   45a <__i686.get_pc_thunk.cx>
  444:  81 c1 b0 1b 00 00       add    ecx,0x1bb0
  44a:  8b 81 f0 ff ff ff       mov    eax,DWORD PTR [ecx-0x10]
  450:  8b 00                   mov    eax,DWORD PTR [eax]
  452:  03 45 08                add    eax,DWORD PTR [ebp+0x8]
  455:  03 45 0c                add    eax,DWORD PTR [ebp+0xc]
  458:  5d                      pop    ebp
  459:  c3                      ret

0000045a <__i686.get_pc_thunk.cx>:
  45a:  8b 0c 24                mov    ecx,DWORD PTR [esp]
  45d:  c3                      ret
int myglob = 42;

int ml_func(int a, int b)
{
    return myglob + a + b;
}

0000043c <ml_func>:

push ebp
mov ebp,esp
call 45a <__i686.get_pc_thunk.cx>
add ecx,0x1bb0
mov eax,DWORD PTR [ecx-0x10]
mov eax,DWORD PTR [eax]
add eax,DWORD PTR [ebp+0x8]
add eax,DWORD PTR [ebp+0xc]
pop ebp
ret

0000045a <__i686.get_pc_thunk.cx>:

mov ecx,DWORD PTR [esp]
ret
int myglob = 42;

int ml_func(int a, int b)
{
    return myglob + a + b;
}

• Save EIP into ECX
int myglob = 42;

int ml_func(int a, int b)
{
    return myglob + a + b;
}

0000043c <ml_func>:
   43c:  55                      push ebp
   43d:  89 e5                   mov ebp,esp
   43f:  e8 16 00 00 00          call 45a <__i686.get_pc_thunk.cx>
   444:  81 c1 b0 1b 00 00       add ecx,0x1bb0
   44a:  8b 81 f0 ff ff ff       mov eax,DWORD PTR [ecx-0x10]
   450:  8b 00                   mov eax,DWORD PTR [eax]
   452:  03 45 08                add eax,DWORD PTR [ebp+0x8]
   455:  03 45 0c                add eax,DWORD PTR [ebp+0xc]
   458:  5d                      pop ebp
   459:  c3                      ret

0000045a <__i686.get_pc_thunk.cx>:
   45a:  8b 0c 24                mov ecx,DWORD PTR [esp]
   45d:  c3                      ret

- Add offset to GOT
- 0x1bb0
```c
int myglob = 42;

int ml_func(int a, int b)
{
    return myglob + a + b;
}
```

**PIC example**

- Access address of a specific GOT entry
- Save it in EAX

```
0000043c <ml_func>:
   43c:  55                      push   ebp
   43d:  89 e5                   mov    ebp,esp
   43f:  e8 16 00 00 00          call   45a <__i686.get_pc_thunk.cx>
   444:  81 c1 b0 1b 00 00       add    ecx,0x1bb0
   44a:  8b 81 f0 ff ff ff       mov    eax,DWORD PTR [ecx-0x10]
   450:  8b 00                   mov    eax,DWORD PTR [eax]
   452:  03 45 08                add    eax,DWORD PTR [ebp+0x8]
   455:  03 45 0c                add    eax,DWORD PTR [ebp+0xc]
   458:  5d                      pop    ebp
   459:  c3                      ret

0000045a <__i686.get_pc_thunk.cx>:
   45a:  8b 0c 24                mov    ecx,DWORD PTR [esp]
   45d:  c3                      ret
```
int myglob = 42;

int ml_func(int a, int b)
{
    return myglob + a + b;
}

0000043c <ml_func>:
  43c:  55                      push   ebp
  43d:  89 e5                   mov    ebp,esp
  43f:  e8 16 00 00 00          call   45a <__i686.get_pc_thunk.cx>
  444:  81 c1 b0 1b 00 00       add    ecx,0x1bb0
  44a:  8b 81 f0 ff ff ff       mov    eax,DWORD PTR [ecx-0x10]
  450:  8b 00                   mov    eax,DWORD PTR [eax]
  452:  03 45 08                add    eax,DWORD PTR [ebp+0x8]
  455:  03 45 0c                add    eax,DWORD PTR [ebp+0xc]
  458:  5d                      pop    ebp
  459:  c3                      ret

0000045a <__i686.get_pc_thunk.cx>:
  45a:  8b 0c 24                mov    ecx,DWORD PTR [esp]
  45d:  c3                      ret

---

**PIC example**

- Load the value of the variable at the address pointed by EAX
- In EAX again
What about function calls?
What about function calls?

- Same approach can work
- But this is not how it is done
Late binding

- When a shared library refers to some function, the real address of that function is not known until load time
  - Resolving this address is called binding
- We can use GOT
  - Same as for variables
Lazy procedure binding

• In large libraries many routines are never called
  • Libc has over 600
    – The number of functions is much larger than the number of global variables
  • It's ok to bind all routines when the program is statically linked
    – Binding is done offline, no runtime cost
• But with dynamic linking run-time overhead is too high
  – Lazy approach, i.e., linking only when used, works better
Procedure linkage table (PLT)

Code:

```
call func@PLT
...
...
```

PLT:

```
PLT[0]:
call resolver
...
PLT[n]:
jmp *GOT[n]
prepare resolver
jmp PLT[0]
```

GOT:

```
...
GOT[n]:
<addr>
```
Procedure linkage table (PLT)

- PLT is part of the executable text section
  - A set of entries
    - A special first entry
    - One for each external function
- Each PLT entry
  - Is a short chunk of executable code
  - Has a corresponding entry in the GOT
    - Contains an actual offset to the function
    - Only after it is resolved by the dynamic loader
Each PLT entry but the first consists of these parts:

- A jump to a location which is specified in a corresponding GOT entry
- Preparation of arguments for a "resolver" routine
- Call to the resolver routine, which resides in the first entry of the PLT
Before function is resolved

- Nth GOT entry points to after the jump
PLT after the function is resolved

- Nth GOT entry points to the actual function
int ml_util_func(int a) {
    return a + 1;
}

int ml_func(int a, int b) {
    int c = b + ml_util_func(a);
    myglob += c;
    return b + myglob;
}

00000477 <ml_func>:
477: 55                       push   ebp
478: 89 e5                   mov    ebp,esp
47a: 53                      push   ebx
47b: 83 ec 24                sub    esp,0x24
47e: e8 e4 ff ff ff          call   467 <__i686.get_pc_thunk.bx>
483: 81 c3 71 1b 00 00       add    ebx,0x1b71
489: 8b 45 08                mov    eax,DWORD PTR [ebp+0x8]
48c: e8 0c ff ff ff          call   3a0 <ml_util_func@plt>
...

000003a0 <ml_util_func@plt>:
3a0: ff a3 14 00 00 00       jmp    DWORD PTR [ebx+0x14]
3a6: 68 10 00 00 00          push   0x10
3ab: e9 c0 ff ff ff          jmp    370 <__init+0x30>
PIC example (functions)

- Resolve the address of GOT
- First learn EIP
  - Saved in EBX
- Then add offset to EBX

```c
int ml_util_func(int a)
{
    return a + 1;
}

int ml_func(int a, int b)
{
    int c = b + ml_util_func(a);
    myglob += c;
    return b + myglob;
}
```
int ml_util_func(int a)
{
    return a + 1;
}

int ml_func(int a, int b)
{
    int c = b + ml_util_func(a);
    myglob += c;
    return b + myglob;
}

00000477 <ml_func>:
  477:  55                      push   ebp
  478:  89 e5                   mov    ebp,esp
  47a:  53                      push   ebx
  47b:  83 ec 24                sub    esp,0x24
  47e:  e8 e4 ff ff ff          call   3a0 <ml_util_func@plt>
  483:  81 c3 71 1b 00 00       add    ebx,0x1b71
  489:  8b 45 08                mov    eax,DWORD PTR [ebp+0x8]
  48c:  e8 0c ff ff ff          call   3a0 <ml_util_func@plt>

000003a0 <ml_util_func@plt>:
  3a0:  ff a3 14 00 00 00       jmp    DWORD PTR [ebx+0x14]
  3a6:  68 10 00 00 00          push   0x10
  3ab:  e9 c0 ff ff ff          jmp    370 <_init+0x30>
PIC: Advantages and disadvantages

• Any ideas?
PIC: Advantages and disadvantages

• Bad
  • Code gets slower
    – One register is wasted to keep GOT pointer
      • x86 has 6 registers, loosing one of them is bad
    – One more memory dereference
      • GOT can be large (lots of global variables)
      • Extra memory dereferences can have a high cost due to cache misses
    – One more call to find GOT
  
• Good
  • Share memory of common libraries
  • Address space randomization
Back to shared libraries
Loading a dynamically linked ELF program

- Map ELF sections into memory
- Note the interpreter section
  - Usually ld.so
- Map ld.so into memory
  - Start ld.so instead of the program
- Linker (ld.so) initializes itself
- Finds the names of shared libraries required by the program
  - DT_NEEDED entries
Finding libraries in the file system

- DT_RPATH symbol
  - Can be linked into a file by a normal linker at link time
- LD_LIBRARY_PATH
- Library cache file
  - /etc/ld.so.conf
  - This is the most normal way to resolve library paths
- Default library path
  - /usr/lib
Loading more libraries

- When the library is found it is loaded into memory
  - Linker adds its symbol table to the linked list of symbol tables
  - Recursively searches if the library depends on other libraries
    - Loads them if needed
Shared library initialization

- Remember PIC needs relocation in the data segment and GOT
  - `ld.so` linker performs this relocation
Conclusion

- Program loading
  - Storage allocation
- Relocation
  - Assign load address to each object file
  - Patch the code
- Symbol resolution
  - Resolve symbols imported from other object files
Thank you!
Weak vs strong symbols

- Virtually every program uses printf
  - Printf can convert floating-point numbers to strings
    - Printf uses fcvt()
  - Does this mean that every program needs to link against floating-point libraries?
- Weak symbols allow symbols to be undefined
  - If program uses floating numbers, it links against the floating-point libraries
    - fcvt() is defined an everything is fine
  - If program doesn't use floating-point libraries
    - fcvt() remains NULL but is never called
#include <stdio.h>

void func_a(void){
    printf("func_a\n");
    return;
}

void func_b(void) {
    printf("func_b\n");
    return;
}

int main(int ac, char **av)
{
    void (*fp)(void);

    fp = func_b;
    fp();
    return;
}
Function pointers

08048432 <func_b>:
push %ebp
mov %esp,%ebp
sub $0x18,%esp
movl $0x8048507,(%esp)
call 80482f0 <puts@plt>
nop
leave
ret

08048447 <main>:
push %ebp
mov %esp,%ebp
and $0xfffffffff0,%esp
sub $0x10,%esp
# Load pointer to func_p on the stack
movl $0x8048432,0xc(%esp)
mov 0xc(%esp),%eax
call %eax
nop
leave
ret
Function pointers

08048432 <func_b>:

```
  8048432: 55    push   %ebp
  8048433: 89 e5  mov    %esp,%ebp
  8048435: 83 ec 18 sub    $0x18,%esp
  8048438: c7 04 24 07 85 04 08 movl   $0x8048507,(%esp)
  804843f: e8 ac fe ff ff call   80482f0 <puts@plt>
  8048444: 90    nop
  8048445: c9    leave
  8048446: c3    ret
```

08048447 <main>:

```
  8048447: 55    push   %ebp
  8048448: 89 e5  mov    %esp,%ebp
  804844a: 83 e4 f0 and    $0xfffffff0,%esp
  804844d: 83 ec 10 sub    $0x10,%esp
  8048450: c7 44 24 0c 32 84 04 movl   $0x8048432,0xc(%esp)
  8048457: 08
  8048458: 8b 44 24 0c mov    0xc(%esp),%eax
  804845c: ff d0 call   *%eax   # Call %eax
  804845e: 90    nop
  804845f: c9    leave
  8048460: c3    ret
```
0804a01c B __bss_start
0804a01c b completed.6591
0804a014 D __data_start
0804a014 W data_start
....
0804a01c D _edata
0804a020 B _end
08048484 T _fini
....
08048294 T _init
....
080483ed T main
...
080482f0 T _start
...