Linking & Loading

HLL source code

preprocessor

->

compiler

assembly code

assembler

machine code

linker

loader

running program
Quick overview of the assembler

Assembler scans the source file(s) top to bottom, dealing with each line-oriented item it finds.

Along the way:
- assigns addresses for everything possible
- each symbol \( \rightarrow \text{addr} \) is entered into the symbol table

In source:

1. **Directives**
   - identifies code or data
   - causes variables to be right size & initial value

2. **Code**

Virtual addresses are used (assigned) in assembly.

Data & code segment each goes at address 0. Separate assembly means we may have several data & code portions that each start at address 0.
From the textbook:

"Linking is the process of collecting and combining various pieces of code and data into a single file that can be loaded (copied) into memory and executed."
Need to consider addresses.
They can be

1. absolute
2. relative
int x = 65;

int main() {
    x++;
    return 0;
}

/* global variable goes in data segment */
add1: file format elf32-i386

Contents of section .data:

8049608 00000000 41000000

08048394 <main>:

8048394: 55  push %ebp
8048395: 89 e5  mov %esp,%ebp
8048397: a1 0c 96 04 08 mov 0x804960c,%eax
804839c: 83 c0 01  add $0x1,%eax
804839f: a3 0c 96 04 08 mov %eax,0x804960c
80483a4: b8 00 00 00 00 mov $0x0,%eax
80483a9: 5d  pop %ebp
80483aa: c3  ret

int x = 65;
int main() {
    int x;
    int a;

    x = 0;
    a = 1;
    while ( x < 300 ) {
        a = a - 12 + x;
        x++;
    }

    return 0;
}
08048394 <main>:

8048394:  55  push %ebp
8048395:  89 e5  mov %esp,%ebp
8048397:  83 ec 10  sub $0x10,%esp
804839a:  c7 45 f8 00 00 00 00  movl $0x0,-0x8(%ebp)
80483a1:  c7 45 fc 01 00 00 00  movl $0x1,-0x4(%ebp)
80483a8:  eb 10  jmp 80483ba

80483aa:  8b 45 fc  mov -0x4(%ebp),%eax
80483ad:  83 e8 0c  sub $0xc,%eax
80483b0:  03 45 f8  add -0x8(%ebp),%eax
80483b3:  89 45 fc  mov %eax,-0x4(%ebp)
80483b6:  83 45 f8 01  addl $0x1,-0x8(%ebp)
80483ba:  81 7d f8 2b 01 00 00  cmpl $0x12b,-0x8(%ebp)
80483c1:  7e e7  jle 80483aa
80483c3:  b8 00 00 00 00  mov $0x0,%eax
80483c8:  c9  leave
80483c9:  c3  ret

op code not an address
Consider this not-quite-complete C code to get at details of linking.

module1.c

```c
int a = 1;
int b = 2;
int main() {
    f1();
}
```

module2.c

```c
int c = 3;
void f1() {
    b = 12;
}
```

assembler cannot compute
1 offset for ‘call f1’
2 absolute address for reference to b

The linker composes
1 data segment
1 code segment
The assembler generates a list of
- type of issue to fix
- location within module to be fixed
- symbol or target needed to make the fix

```c
for main() & f1();
push %ebp  55
movl %esp, %ebp 89 e5
call f1
.L2:
```

Symbol table
- main 0
- .L2 8

List where: bytes 4-7
what: offset
needed: addr for f1

```c
```
for \ f1() \& 3

push %ebp 550
movl %esp,%ebp 891e5
movl $12, b c705

symbol table
\f1 \ 0

List:
where: bytes 5-8
what: absolute address
needed: addr of b

immed 12
data segment for module1.c

symbol table

<table>
<thead>
<tr>
<th>a</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>4</td>
</tr>
</tbody>
</table>

data segment for module2.c

symbol table

<table>
<thead>
<tr>
<th>c</th>
<th>0</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>a</th>
<th>01</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>00</td>
</tr>
<tr>
<td></td>
<td>00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>b</th>
<th>02</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>00</td>
</tr>
<tr>
<td></td>
<td>00</td>
</tr>
<tr>
<td></td>
<td>00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>c</th>
<th>03</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>00</td>
</tr>
<tr>
<td></td>
<td>00</td>
</tr>
<tr>
<td></td>
<td>00</td>
</tr>
</tbody>
</table>
The linker receives memory image(s) (as complete as possible) symbol table(s) List(s) called relocation.

Combines it into

\[
\begin{align*}
\text{module 1 code} & \quad \text{contiguous segment} \\
\text{module 2 code} & \\
\text{module 1 data} & \quad \text{contiguous segment} \\
\text{module 2 data} & \\
\end{align*}
\]

Let module 1 be first for ordering:

\[
\begin{array}{c}
\text{module 1 code} \\
\text{module 2 code}
\end{array}
\]

Within VA space, 32-bit Linux, base address is \(0x08048000\)

Pretend module 1 size is 30 bytes

module 2 size is 20 bytes

30+20 bytes
Data segment follows code segment within VA space, aligned at a 4K byte boundary.

0x08049000

segment size = 12 bytes

The ordering of the variety of segments will be fixed and programmed into the linker.

See section 7.4, page 658 for set of segments and what each is for.
To complete module 1 code:

Figure out bytes 4..7 (offset from code base address)

\[ 0x08048004 \]
\[ 0x08048005 \]
\[ 0x08048006 \]
\[ 0x08048007 \] an offset

(target)

\[ f1 \text{ is at } 0x08048000 + 1e \]
\[ = 0x0804801e \]

Offset will be (more or less) \( \text{target} - .L2 \)

\[ 0x0804801e - 0x08048008 \]

8 + module 1 base addr

\[ 3010 \]
To complete module 2 code:
need absolute address of b to be placed in bytes 5-8 of module 2 code.

\[
\text{absolute addr of } b = \text{ module 1 data segment + offset within segment base address } = 0x08049000 + 4
\]
Now we have produced an **executable**

Specialized form for a given platform

---

OS + architecture

<table>
<thead>
<tr>
<th>platform</th>
</tr>
</thead>
<tbody>
<tr>
<td>entry point (initial PC)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>code segment</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>data segment</th>
</tr>
</thead>
</table>

| symbol table + debugging info |
Loading
(simplified, to get the idea)

- copy executable into memory
- allocate space for stack and set %esp
- allocate space for heap
- set PC (jump to entry point)