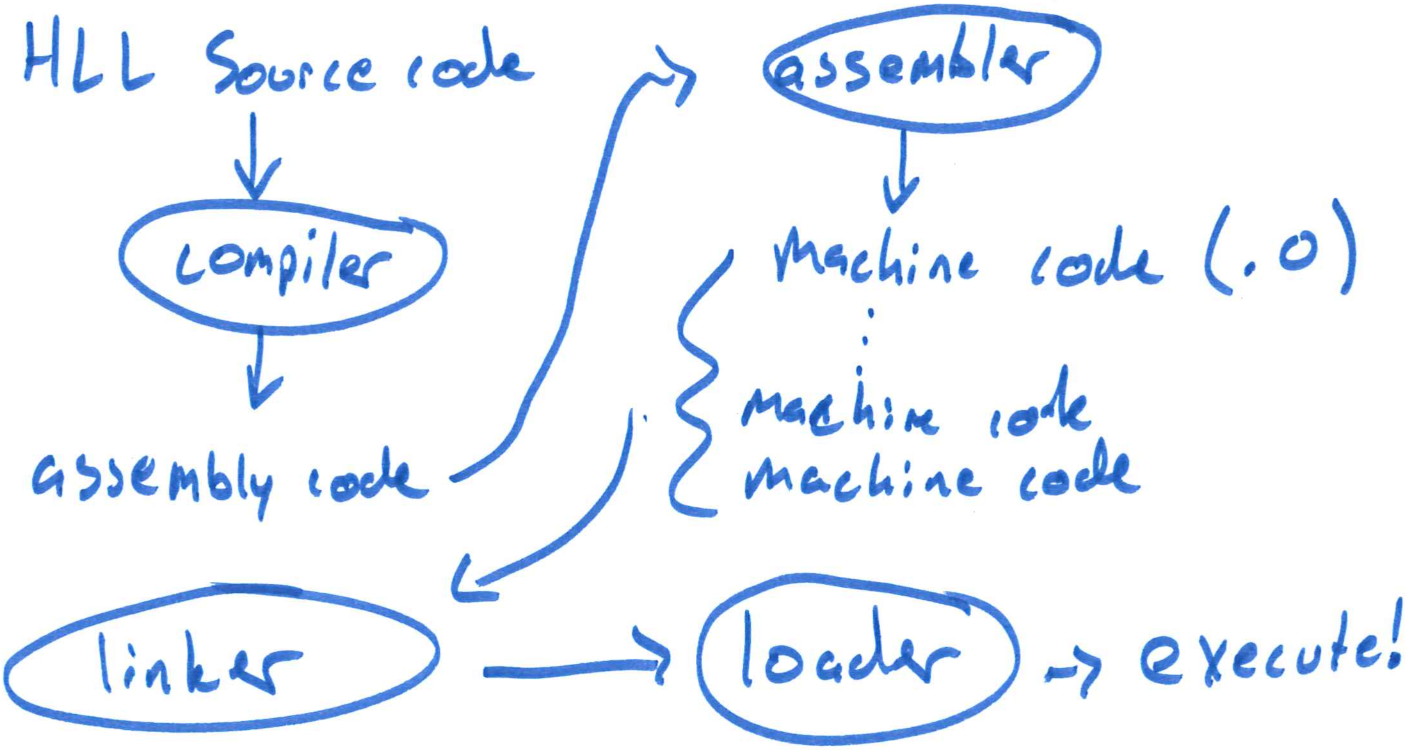
Linking and loading



Difference between assembly + machine code?

- Why do we need the assembler?

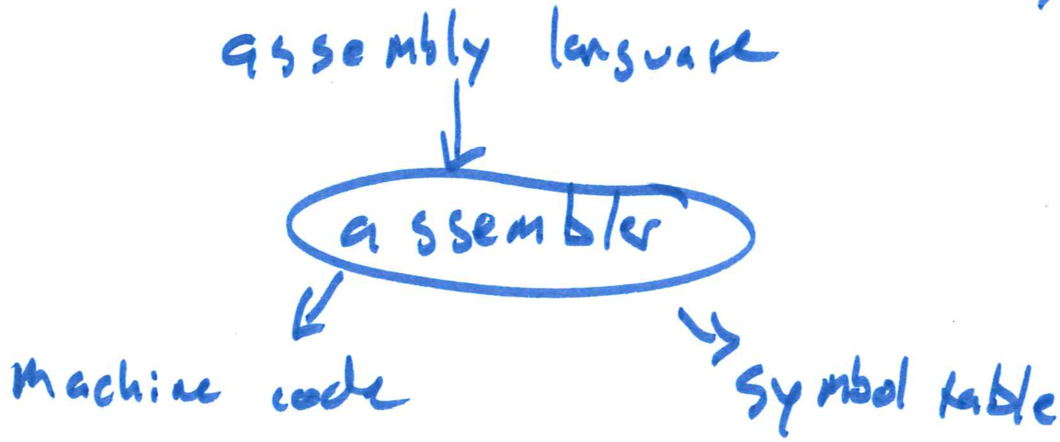
```
top:  add ...
      movl ....
      jmp top
```

address

- convert from assembly to binary
- assign addresses to <u>symbols</u>

Assembler:

- Scan each line and convert to machine code
- assign each symbol (it can) an <u>address</u> → virtual address
- these symbol → address "translations" are put in a symbol table



objdump -t

After assembling ~~the~~ into machine code

"Linking" - collects and combines machine code and data into a file that is

"Loaded" - copied into memory to be ready to <u>execute</u>

Addressing in machine code

- absolute - the actual (virtual) address
- relative - save space use PC + offset

(Indirect)

```
movl 0x5000000, %eax
```

```
while (...) {
    ...
    ...
}
```

jmp ~~0x80123AB~~ - 10

---

need a linker to compute these addresses

Mod1.c

```
int a = 12;
main () {
    f1();
}
```

(f1) → assembler doesn't know address

Mod2.c

```
f1 () {
    a++;
}
```

(a++) → assembler can't know what address this is

later the linker fills in the missing addresses

```
int a = 12;

int main(int argc, char *argv[]) {

  for (int i=0; i<10; i++)  {
    a += 1;
  }

  return 0;
}
```

global var

```
addressing:     file format elf32-i386

Disassembly of section .text:

08048394 <main>:
 8048394: 55                        push   %ebp
 8048395: 89 e5                     mov    %esp,%ebp
 8048397: 83 ec 10                  sub    $0x10,%esp
 804839a: c7 45 fc 00 00 00 00      movl   $0x0,-0x4(%ebp)
 80483a1: eb 11                     jmp    80483b4 <main+0x20>
 80483a3: a1 2c 96 04 08            mov    0x804962c,%eax
 80483a8: 83 c0 01                  add    $0x1,%eax
 80483ab: a3 2c 96 04 08            mov    %eax,0x804962c
 80483b0: 83 45 fc 01               addl   $0x1,-0x4(%ebp)
 80483b4: 83 7d fc 09               cmpl   $0x9,-0x4(%ebp)
 80483b8: 7e e9      -17            jle    80483a3 <main+0xf>
 80483ba: b8 00 00 00 00            mov    $0x0,%eax
 80483bf: c9                        leave

Disassembly of section .data:

08049628 <__data_start>:
 8049628: 00 00                     add    %al,(%eax)
  ...

0804962c <a>:
 804962c: 0c 00                     or     $0x0,%al
  ...
```

gcc -c → only do preprocessor + compiler + assembler

↳ .o file

ld → linker → links .o files together

cpp (preprocessor)
cc (compiler)
as (assembler)
ld (linker)

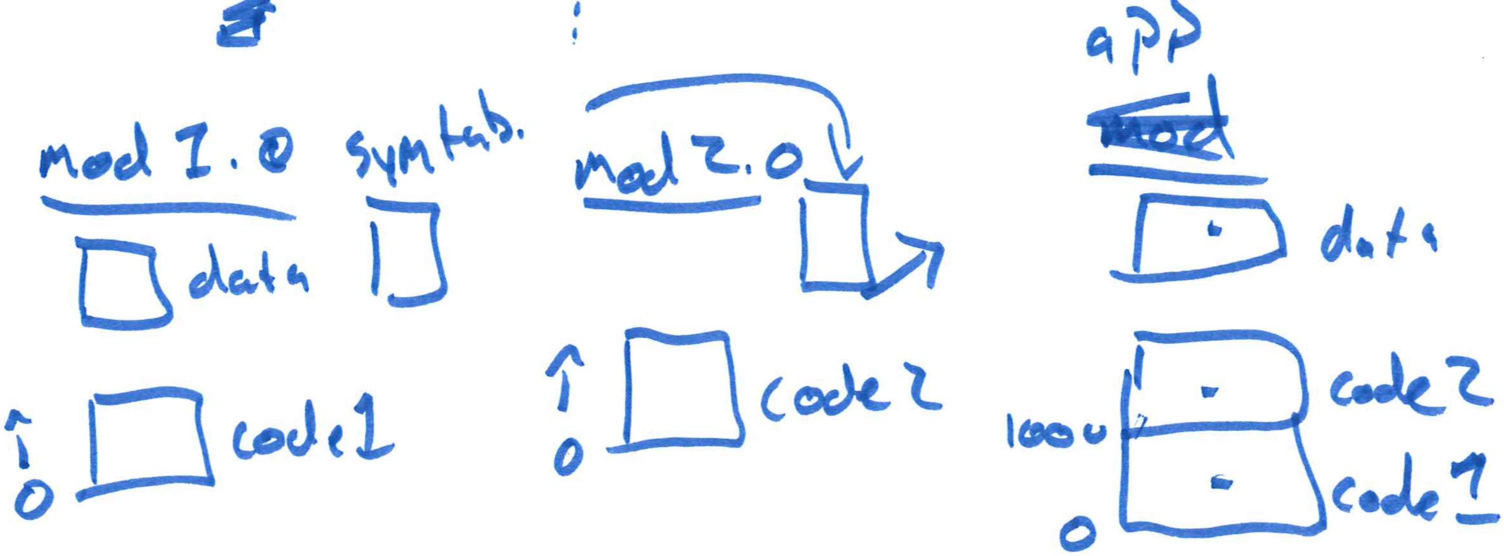
gcc → all these together

Linker fills in other missing info too

Creates the ELF → file format for executables

→ 15+ segments
.data .rodata
.text .bss
⋮

adds .init:
.exit:



fills in blanks, assigns addresses

- relocate ~~o~~ object files
  update any addresses of moved objects

Loading

- done (mostly) by the OS
- copy executable (created by linking) into memory
- OS allocates some space for stack
  - sets %esp
- OS allocates some space for heap
  - sets brk pointer
- sets the PC to first instruction
  ↳ starts executing code

  logically jumps to main.
  actually goes to init() ↗ initializes C libraries

Previously, we assumed <u>Static</u> linking

<u>Dynamic</u> linking common

↳ parts of the linking step saved until execution / loading

Why dynamic?

- two programs both using same func.
  - save memory by having 1 copy
- lots of code never needed
- reduces executable size
- eliminates duplication in executables
- to dynamically choose which function