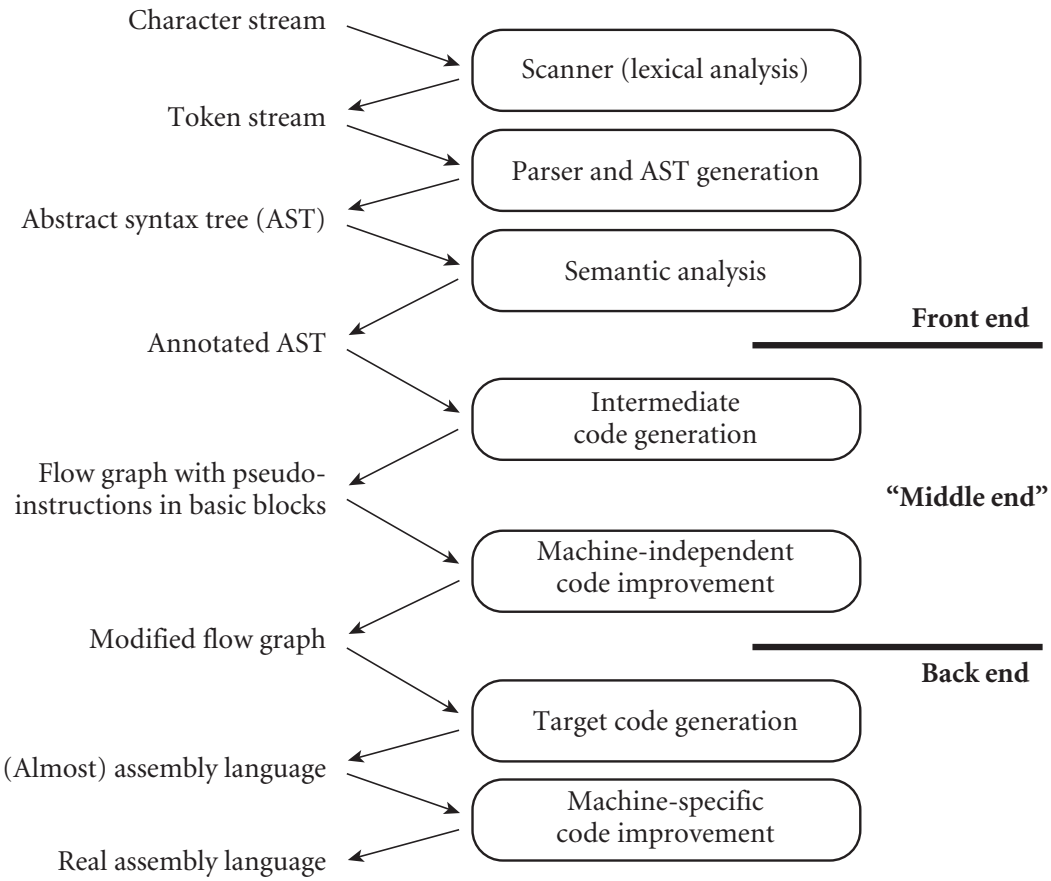
Notes for CSC 254, 24 Apr. 2025

==========

Recall compiler phases:



Note that this differs slightly from the version shown in the Chapter 1 lectures. Specifically, I've

- rolled AST generation into the parser
- separated semantic analysis from (medium level) IF generation
- put that IF generation in the "middle end"

This more accurately reflects the likely structure of a modern compiler.

It's common for a compiler to have more than one intermediate form/representation (IF/IR). These are sometimes differentiated by "level," or degree of abstractness:

high-level—typically an AST

medium-level—often a ***control flow graph***
basic blocks as nodes; jumps as edges
low-level—usually instructions for an idealized machine
perhaps the same notation that's used w/in basic blocks above

NB: there are no hard boundaries between these levels.

One family of IFs deserves separate mention: ***stack-based IFs***
may be medium or low-level
not used in most compilers, but important in special cases
particularly where size is an issue
examples include JBC, CIL, 1970s pcode

example from the book: Heron's formula:

```
A = sqrt [s(s-a)(s-b)(s-c)]
where s = (a+b+c)/2
```

```
stack-based:        3-address pseudo-assembly
   push a              r2 := a
   push b              r3 := b
   push c              r4 := c
   add                 r1 := r2 + r3
   add                 r1 := r1 + r4
   push 2              r1 := r1 / 2        -- s
   divide
   pop s
   push s
   push s              r2 := r1 - r2       -- s-a
   push a
   subtract
   push s              r3 := r1 - r3       -- s-b
   push b
   subtract
   push s              r4 := r1 - r4       -- s-c
   push c
   subtract
   multiply            r3 := r3 * r4
   multiply            r2 := r2 * r3
   multiply            r1 := r1 * r2
   push sqrt           call sqrt
   call
```

time-space tradeoff

stack code is denser

lots of instructions, but tiny

v. speed

can't optimize for register set and pipeline performance

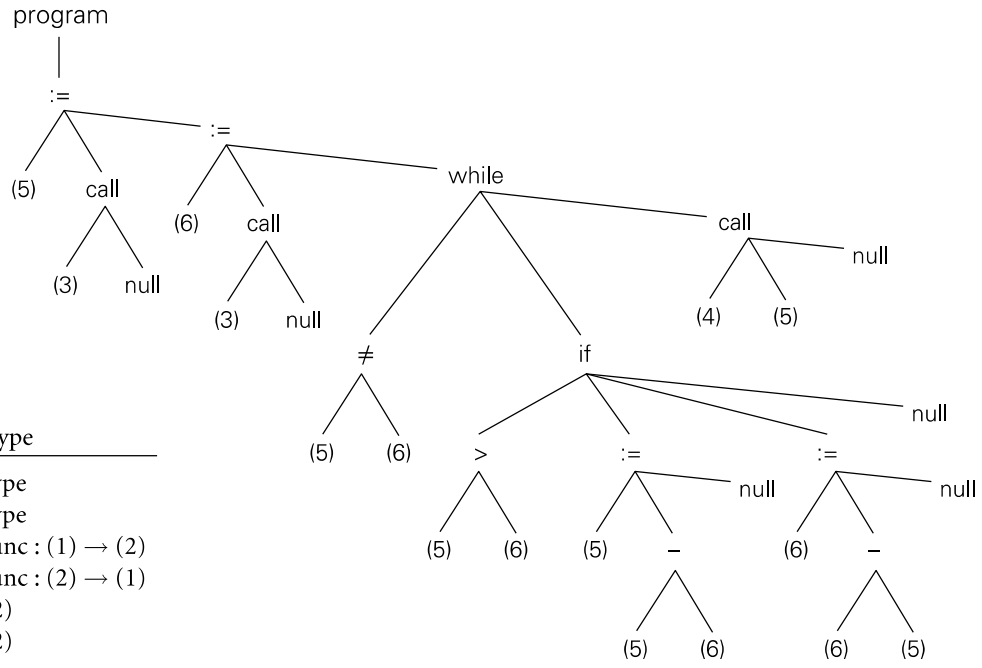
The JBC or CIL version of the stack-based code will use a single byte for every instruction except the second-to-last, which will take 3 bytes. That's 23 instructions in 25 bytes.

The 3-address code keeps a, b, c, and s in registers, and uses only 13 instructions. Typically, however, most will be 4 bytes long (the last will be 8). That's 13 instructions in 56 bytes.

Consider the GCD example from the Chap. 1 of the book. Source (in C):

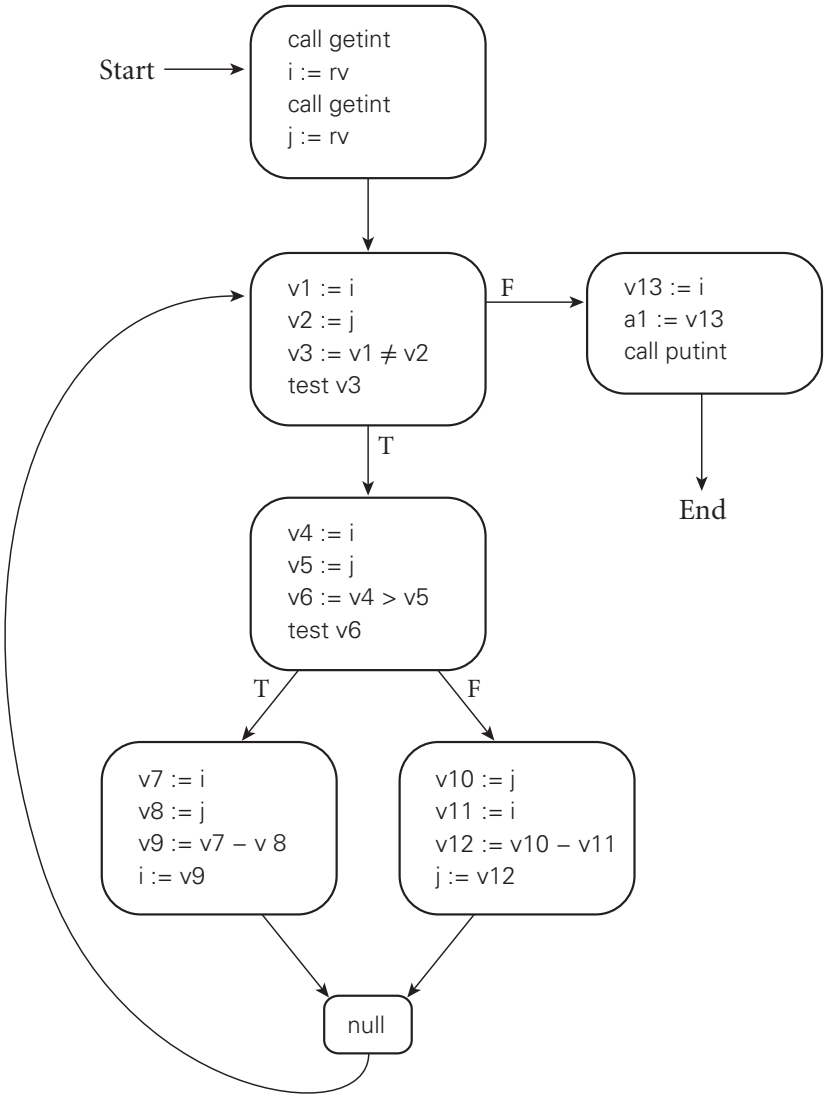
```
int main() {
   int i = getint(), j = getint();
   while (i != j) {
      if (i > j) i = i - j;
      else j = j - i;
   }
   putint(i);
}
```

AST (we know how to generate this now):



Index	Symbol	Type
1	`void`	type
2	`int`	type
3	`getint`	func : (1) → (2)
4	`putint`	func : (2) → (1)
5	`i`	(2)
6	`j`	(2)

Control Flow Graph is “straightforward” to generate from the AST:



Here I’ve used “virtual registers” for all computed values.

- These are assumed to be unlimited in number.
- I’ve also used special register names (a1 and rv) to pass values to and from subroutines.

Conversion from AST to control-flow graph (or other IF) typically uses one or more pass(es) over the tree.

- Like static semantic checking, these pass(es) can be expressed with an AG, with attributes for control flow graph fragments.
- More commonly, it’s just hand-written code.

The control-flow graph may see many changes during code improvement. We may split and merge basic blocks; add and delete blocks; change the code inside blocks; move code from one block to another; etc.

Much of the decision making is driven by ***data flow analysis***, which discovers properties of blocks that depend on other blocks. E.g.,

- which virtual registers are ***live*** (contain values that may be needed in the future at the end of a given block?
- which values are known to be ***available*** (contained in some virtual register) at the start of a given block?

Like the algorithm that builds predict sets for a top-down parser, the data flow "engine" begins with "obvious" facts and iterates until it can't learn anything more (and we can prove the answer has converged).

Conversion to low-level IF can be as simple as picking an order for the basic blocks of the control flow graph:

```
        call getint
        i := rv
        call getint
        j := rv

   L1:  v1 := i
        v2 := j
        v3 := v1 != v2
        test v3
        if false goto L2

        v4 := i
        v5 := j
        v6 := v4 > v5
        test v6
        if false goto L3

        v7 := i
        v8 := j
        v9 := v7 - v8
        i := v9
        goto L4
```

```
L3: v10 := j
    v11 := i
    v12 := v10 – v11
    j := v12

L4: goto L1

L5: v13 := i
    a1 := v13
    call putint
    halt
```

This is unlikely to give you the best code for a given target instruction set; more on this below.

Key tasks of target code generation

instruction selection

This seems like it ought to be straightforward, but it can be tricky
- more than one way to do things on many machines
  - multiply by 2 v. add to self v. left shift one bit
- messy addressing modes
- side effects (e.g., on condition codes or scratch registers)

Common to make a simple choice,
- then follow up w/ machine-dependent code improvement

Both simple choice and improvement may be based on automated
- pattern matching (code generator generator)

instruction scheduling

order in which to execute logically independent instructions

e.g.

```
r2 += r3 * r4   \
r1 := a         / swap these (to tolerate load delay)
r2 += r1
```

register allocation

what should be kept in registers when?
- NP hard in the general case—equivalent to minimal graph coloring
- typical modern compilers use a heuristic solution to
  - the coloring problem

instruction scheduling and register allocation interact in complicated ways
- if you reorder instructions, the number of registers needed may change
  - (have to hang onto a temporary value across the creation of some other temporary value)
- if you run out of registers, you have to ***spill*** them,
  - which changes the set of instructions
  - and the new instructions are loads and stores, for which scheduling is particularly important
- real compiler might
  - schedule instructions assuming unlimited registers
  - allocate registers, spilling as necessary
    - (this is the NP hard graph coloring problem)
    - typical modern compilers use a heuristic solution
  - reschedule to fill new load delays, so long as it doesn't mess up register allocation
- more on this in Chap 17 (not covered this semester)

FWIW, with aggressive (machine independent *and* machine independent) code improvement, even something as simple as the GCD program can produce surprisingly clever code. The following is from LLVM -O3, hand translated from x86-64 assembly back into pseudocode for readability.

```
int main() {
   int i = getint(), j = getint();
   while (i != j) {
      if (i > j) i = i - j;
      else j = j - i;       }
   putint(i);
}
```

```
    call getint
    r1 := rv                 // r1 holds i
    call getint              // rv holds j
    compare rv, r1           // compare j to i
    goto L2 if equal
    r2 := 0
    compare r1, rv
L1:
    r3 := 0
    r3 := rv if less than    // "conditional move"
                             // based on most recent comparison
```

```
    r4 := r1                 // i
    r4 := r2 if less than
    r1 -:= r3                // i -= (i < j ? j : 0)
    rv -:= r4                // j -= (i < j ? 0 : i)
    compare r1, rv
    goto L1 if not equal
L2:
    a1 := r1                 // i
    call putint
```

The inner loop here is only 8 instructions long, compared to 20 in our naive linearized control flow graph.

==========

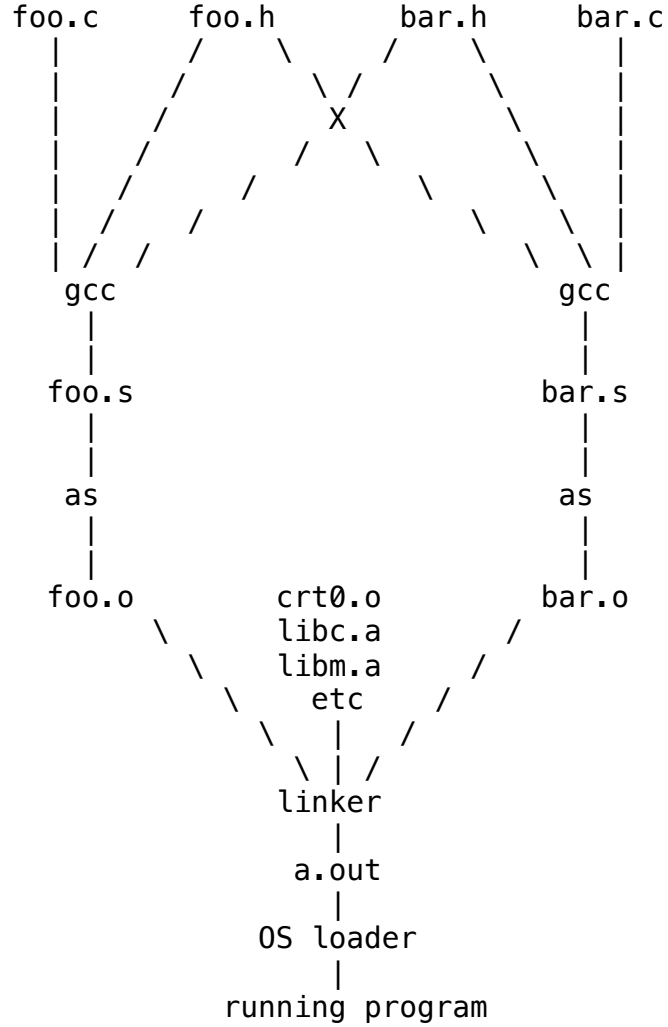
Building a program:

Terminology:

- An ***object file*** contains machine language code and data.
- A ***relocatable*** object file contains the information needed to relocate the file's contents.
- An ***executable*** object file can be loaded and run. (Rust, for some reason, calls object files ***crates***.)

It is possible for a file to be both relocatable and executable.

Example of a C program with 4 source files, `foo.c`, `foo.h`, `bar.c`, and `bar.h`:



Assemblers

Translate assembly language to machine language.
Long ago, had lots of fancy features (e.g., sophisticated macro systems) for the convenience of human users. Nowadays very little assembly code is written by hand.

Some produce assembly code and make the assembler a separate pass.

Some compilers produce machine code directly.

For these you either need an option to produce assembly code on demand or a good disassembler for people developing/debugging the compiler.

Principal complication of assembly is the fact that a label may be used before it is defined:

```
        cmp   %eax, %ecx
        jne   .L1
...
.L1:    addl  %eax, %edx
```

When the assembler sees `jne` (jump if not equal) the first time, it doesn't know `.L1`'s location.

Translation therefore takes 2 steps:

1) associate memory locations with labels, based on an understanding of how long each eventual code block will be (this can be complicated by the fact that the length of some instructions [e.g., branches, loads] depends on how far away things end up).
2) go back and do the actual assembly-to-machine code translation, using the locations figured out in step 1.

Step 2 also generates a symbol table.

Each entry contains

- the string representing the symbol
- the segment—e.g. undefined, absolute, text, data, bss (bss = zero-initialized globals ["block started by symbol"])
- the offset from the start of the segment
- a bit for private versus global

- for symbols not defined here, a list of the instructions in which the symbol is referenced (so the linker can patch them up [see below])

This is in addition to (or an augmentation of) the symbol table produced by the compiler.

Linking

Assemblers (and compilers) seldom produce exactly the bits that will be in the code segment in memory when your program runs. Two tasks generally remain to be done

(1) ***Symbol resolution***

Most programs are made of separately compiled modules.

Something needs to stitch these together to make a whole program. This is called ***linking***; it's done by a ***linker***.

The '.o' files that the assembler produces from your source files are called ***object*** files because they contain "object" code (as opposed to source code). They define certain ***symbols*** that represent interesting things in your program—mainly code and data—and contain ***unresolved references*** to symbols in other object files.

The linker takes a collection of object files and resolves mutual references. It usually knows about certain "standard" libraries that contain many of the symbols.

(2) ***Relocation***

Because your program is typically made from separately-compiled pieces, the assembler doesn't know when it creates a given .o file where in your address space that file will lie. This means it doesn't know the absolute addresses at which code and data will lie.

Branches can be made in terms of relative offsets from the program counter, but jumps, loads, and stores have to be deferred until we know what the absolute address of the beginning of the object file will be.

Once we know this address, we can ***relocate*** the code. This job is usually also done by the linker.

Object files contain information indicating that certain words need to be modified to reflect where symbols have been placed.

– Might be as simple as adding the address of a file to the word
– Or adding some piece of the address to some piece of the word; more on this below.

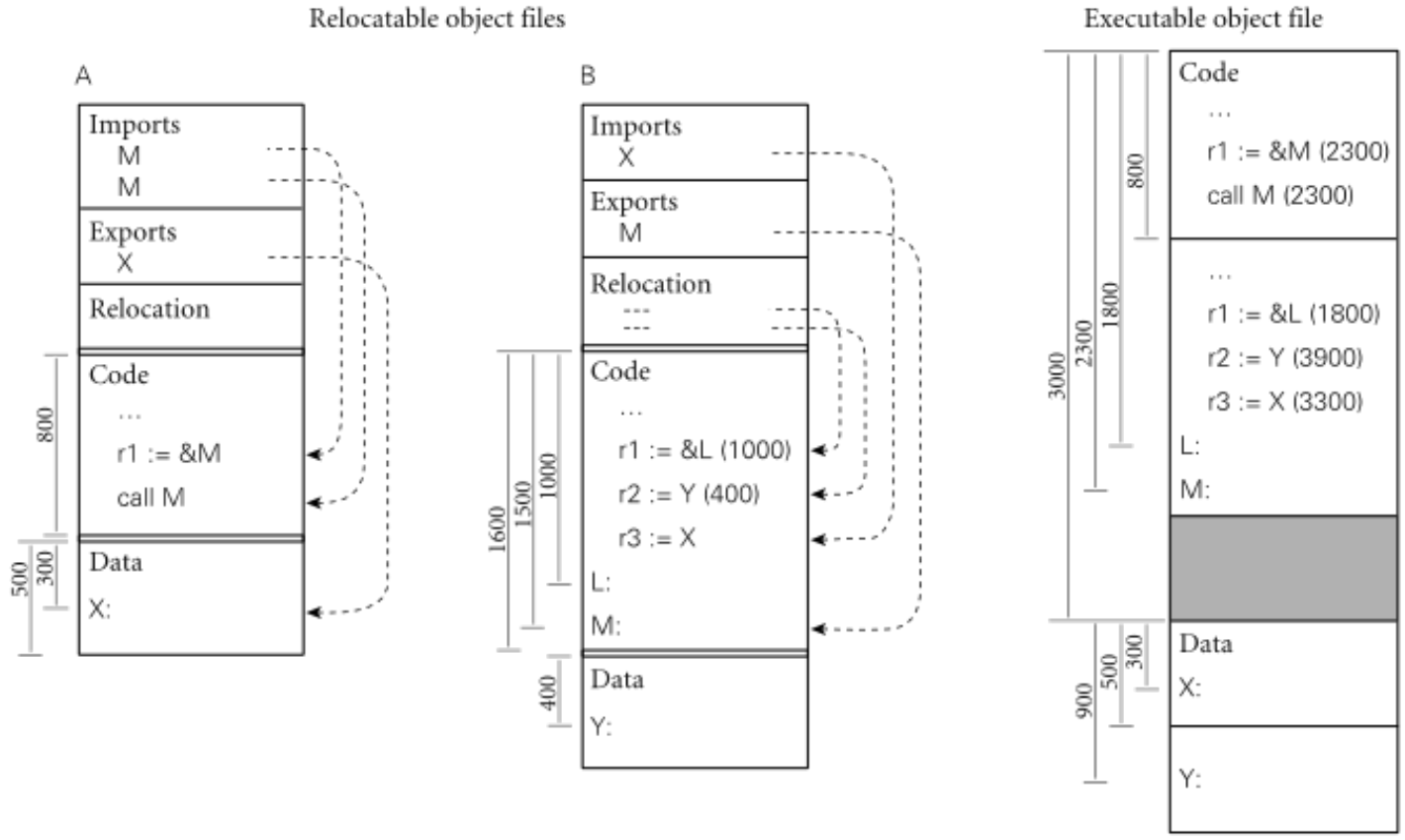


Figure 15.9 **Linking relocatable object files A and B to make an executable object file.** For simplicity of presentation, A's code section has been placed at offset 0, with B's code section immediately after, at offset 800 (addresses increase down the page). To allow the operating system to establish different protections for the code and data segments, A's data section has been placed at the next page boundary (offset 3000), with B's data section immediately after (offset 3500). External references to M and X have been set to use the appropriate addresses. Internal references to L and Y have been updated by adding in the starting addresses of B's code and data sections, respectively.

A warning: the term ***loading*** is sometimes used for relocation. It is better used for the task of putting a program (or at least part of it) into physical memory so it can run. The kernel does loading in response to an exec system call (or its equivalent in non-Unix systems). Once upon a time, when hardware didn't do address translation, programs had to be relocated when they were loaded; hence the confusion. It's especially unfortunate that Unix's linker is called "ld", which suggests "loader". Sometimes a linker is called a "link editor" or (unfortunately) "link-loader".

AND... Just to make life more confusing, modern systems often employ Address Space Layout Randomization (ASLR) as a security measure. This effectively puts relocation ***back*** into the loader's job description.

Unix ELF Object File Format (Executable and Linking Format)

Contains

- ELF header (contains pointer to section header table)
- sections
  - .text — code
  - .rodata — constants
  - .data — initialized, writable data
  - .bss — placeholder for uninitialized data
  - .symtab — global symbols, defined and undefined
  - .rel.text — relocation information for code
  - .rel.data — relocation information for data
  - .debug — debugger symbol table if compiled –g
  - .line — line number map if compiled –g
  - .strtab — heap for strings in .symtab and .debug
- section header table

ELF File Header

from /usr/include/sys/elf.h :

```
typedef struct
{
  unsigned char e_ident[EI_NIDENT]; /* Magic number and other info */
  Elf32_Half e_type;                /* Object file type */
  Elf32_Half e_machine;             /* Architecture */
  Elf32_Word    e_version;          /* Object file version */
  Elf32_Addr    e_entry;            /* Entry point virtual address */
  Elf32_Off     e_phoff;            /* Program header table file offset */
  Elf32_Off     e_shoff;            /* Section header table file offset */
  Elf32_Word    e_flags;            /* Processor-specific flags */
```

```
  Elf32_Half e_ehsize;          /* ELF header size in bytes */
  Elf32_Half e_phentsize;       /* Program header table entry size */
  Elf32_Half e_phnum;           /* Program header table entry count */
  Elf32_Half e_shentsize;       /* Section header table entry size */
  Elf32_Half e_shnum;           /* Section header table entry count */
  Elf32_Half e_shstrndx;        /* Section header string table index */
} Elf32_Ehdr;
```

Details of the relocation information vary from machine to machine.
ELF defines 11 different encodings.
Two of them cover most cases on the x86:

- PC relative branches
  - linker should subtract address of instruction from target address and then add result into field (usually -4)
- absolute jumps
  - linker should add target address into field (usually zero)

RISC machines tend to be quite a bit trickier.
For example, a source statement like

```
void() *f = &foo;
```

is likely to become a PAIR of instructions on even a 32-bit RISC machine:

```
lui r1, c1          # &foo >> 16
ori r1, c2          # &foo & 0xffff
```

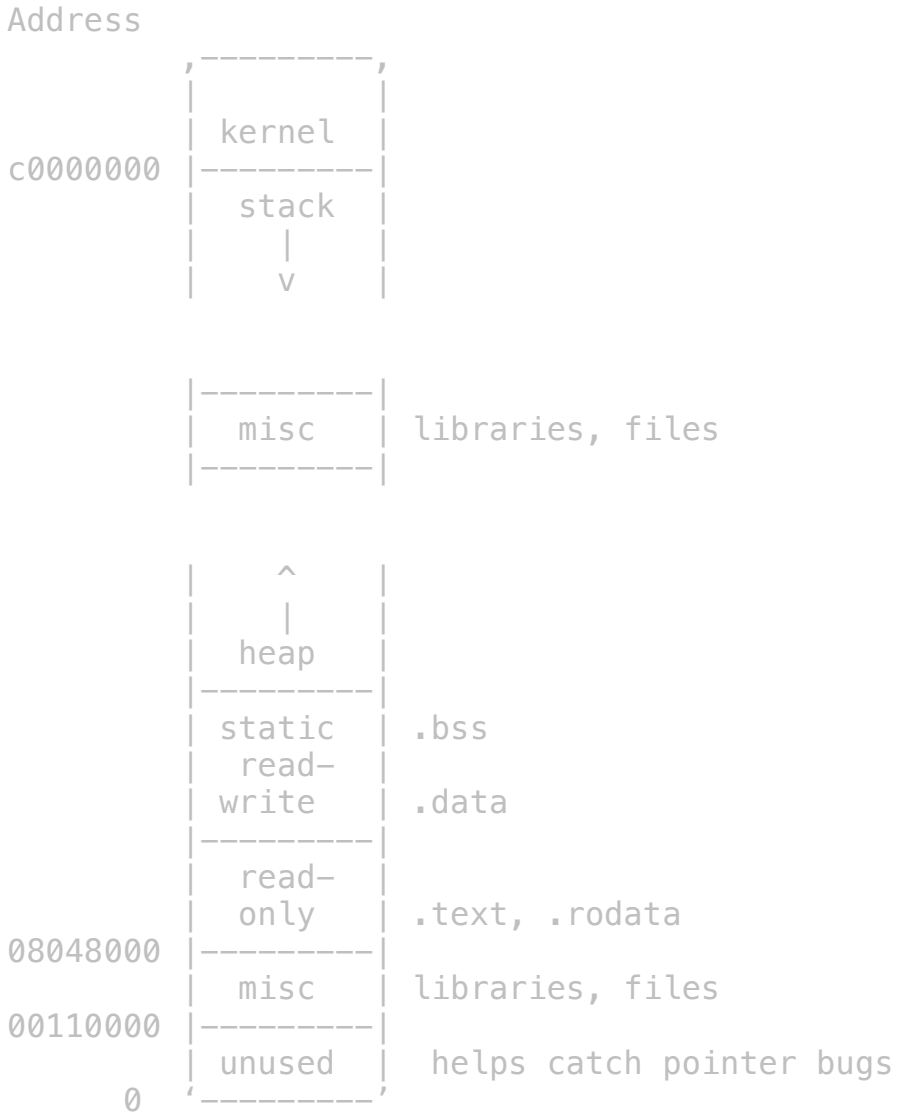
The linker needs to know how to create the two specified constants, given the address of foo, and how to embed them in the immediate fields of the instructions.

Loader: Loads file from disk/secondary storage

- Read header for size of text and data segments
- Create new address space - text, data, stack
- Copy instructions/data from file into new address space (memory)
- Copy program arguments onto stack

Initialize machine registers/stack pointer
Jump to startup routine
- call any static initializers
- copy program arguments from stack to registers (on RISC machine)
- call program's main routine
- on return, terminate program with exit system call

32-bit Linux Memory Layout (slightly updated from the version in the book).
Note: fig is not to scale—kernel occupies 1/4 of address space.



Tools: Several tools can be used to read/interpret object files:

- `od`—displays the contents of any file
- `nm`—displays the symbol table information appended to an object file
- `objdump` (on Linux)
- `readelf` (on Linux)
- `llvm-dwarfdump`

(abbreviated) example

```
% nm -p -v time_test.o

time_test.o:
0000000000 f time_test.c
0000000000 U exit
0000000000 U random
0000000000 U printf
0000000004 D counter
0000000008 D nthreads
0000000044 d count
0000000048 d sense
0000002712 T main
0000003692 T barrier
0000003852 T initialize
0000136824 B t1
0000136828 B t2
0000136832 B t3
```

Key:

- u undefined (external)
- t text (code)
- d initialized data
- b bss
- s section boundary
- f source file boundary
- a absolute (non-relocatable) value

Capital letter means exported global.

Shared libraries

Motivation

- save disk space—don't have copies of libraries in every executable on the disk
- save space in main memory—don't have copies of libraries in every running process in memory
- allow upgrades of libraries without re-compilation—when you replace the shared copy of the library you automatically upgrade every application that is set up to use it (at least the next time it is launched)

Implementation is kind of complicated. Key ideas include

- ***position-independent code*** (PIC)
- linkage tables (for absolute jumps, references to external symbols)
- initialization of tables with `ld.so` address, for lazy code linking

Lots of wrinkles may be different on different systems. For example: x86-32 doesn't allow direct reads of PC (rip); need to fake with call instruction.