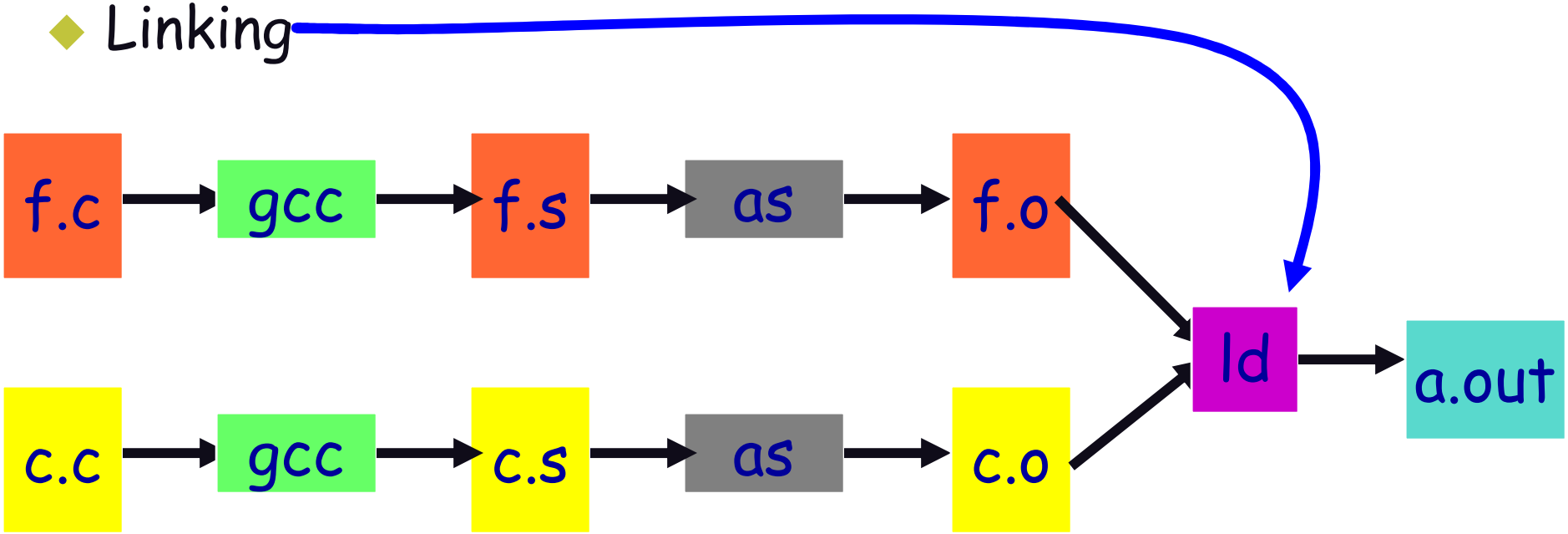
Lecture 8: linking
CS 140

Dawson Engler
Stanford CS department

Today's Big Adventure

- Linking



how to name and refer to things that don't exist yet

how to merge separate name spaces into a cohesive whole

- Readings

man a.out & elf

run "nm" on a few .o and a.out files.

Linking as our first naming system

- Naming = very deep theme that comes up everywhere

  Naming system: maps names to values

- Examples:

  Linking: Where is printf? How to refer to it? How to deal with synonyms? What if it doesn't exist

  Virtual memory address (name) resolved to physical address (value) using page table

  file systems: translating file and directory names to disk locations, organizing names so you can navigate, …

  mit.edu resolved to 18.26.0.1 using DNS table street

  names: translating (elk, pine, …) vs (1st, 2nd, …) to actual location

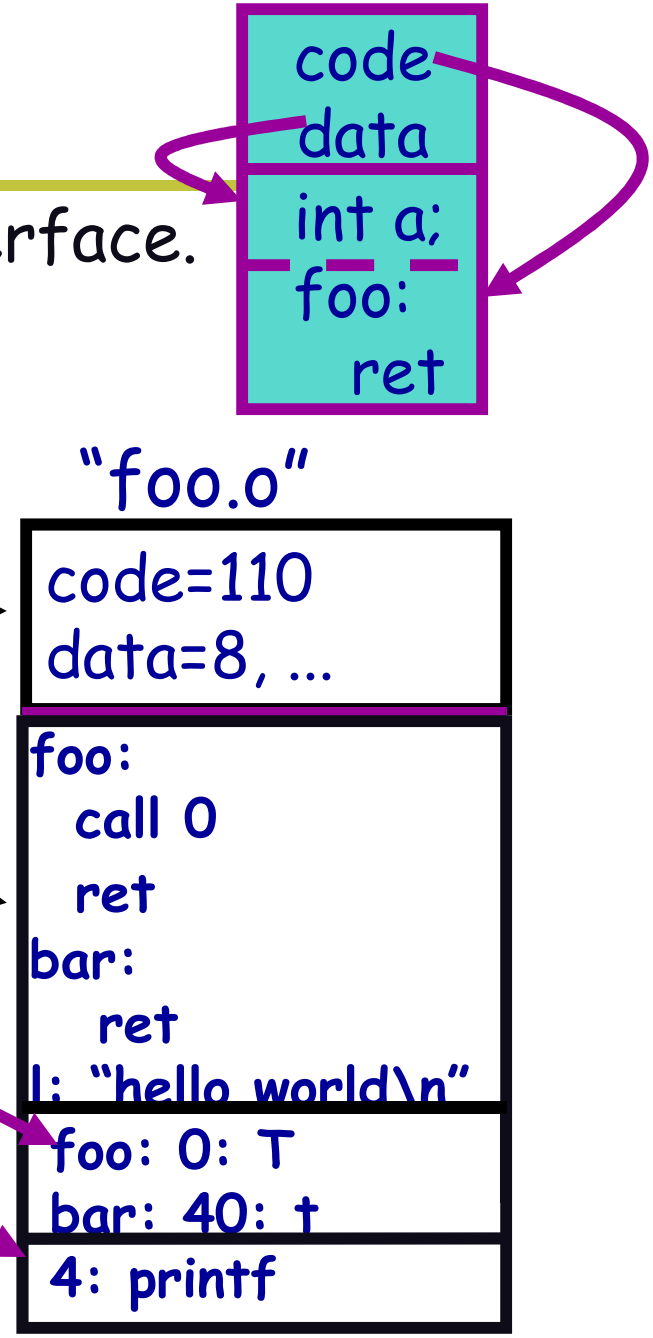
  your name resolved to grade (value) using spreadsheet

Perspectives on information in memory

- Programming language view: add r1, r2, 1
  - instructions: specify operations to perform
  - variables: operands that can change over time
  - constants: operands that never change
- Changeability view (for sharing):
  - read only: code, constants (1 copy for all processes)
  - read and write: variables (each process gets own copy)
- Addresses versus data
  - addresses used to locate something: if you move it, must update address
  - examples: linkers, garbage collectors, changing apartment
- Binding time: when is a value determined/computed?
  - Early to late: compile time, link time, load time, runtime

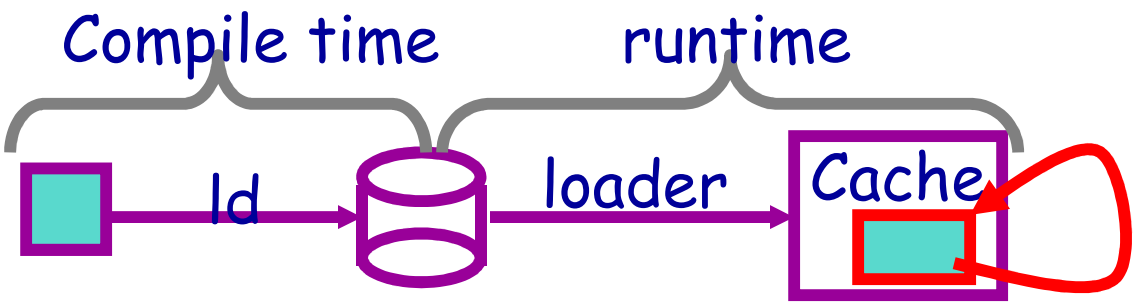
How is a process specified?

- Executable file: the linker/OS interface.
  - What is code? What is data?
  - Where should they live?
- Linker builds ex from object files:
  - Header: code/data size, symtab offset
  - Object code: instructions and data gen'd by compiler
  - Symbol table:
    - external defs (exported objects in file)
    - external refs (global syms used in file)



How is a process created?

- On Unix systems, read by "loader"



  reads all code/data segs into buffer cache; maps code (read only) and initialized data (r/w) into addr space

  fakes process state to look like switched out

- Big optimization fun:

  Zero-initialized data does not need to be read in.

  Demand load: wait until code used before get from disk

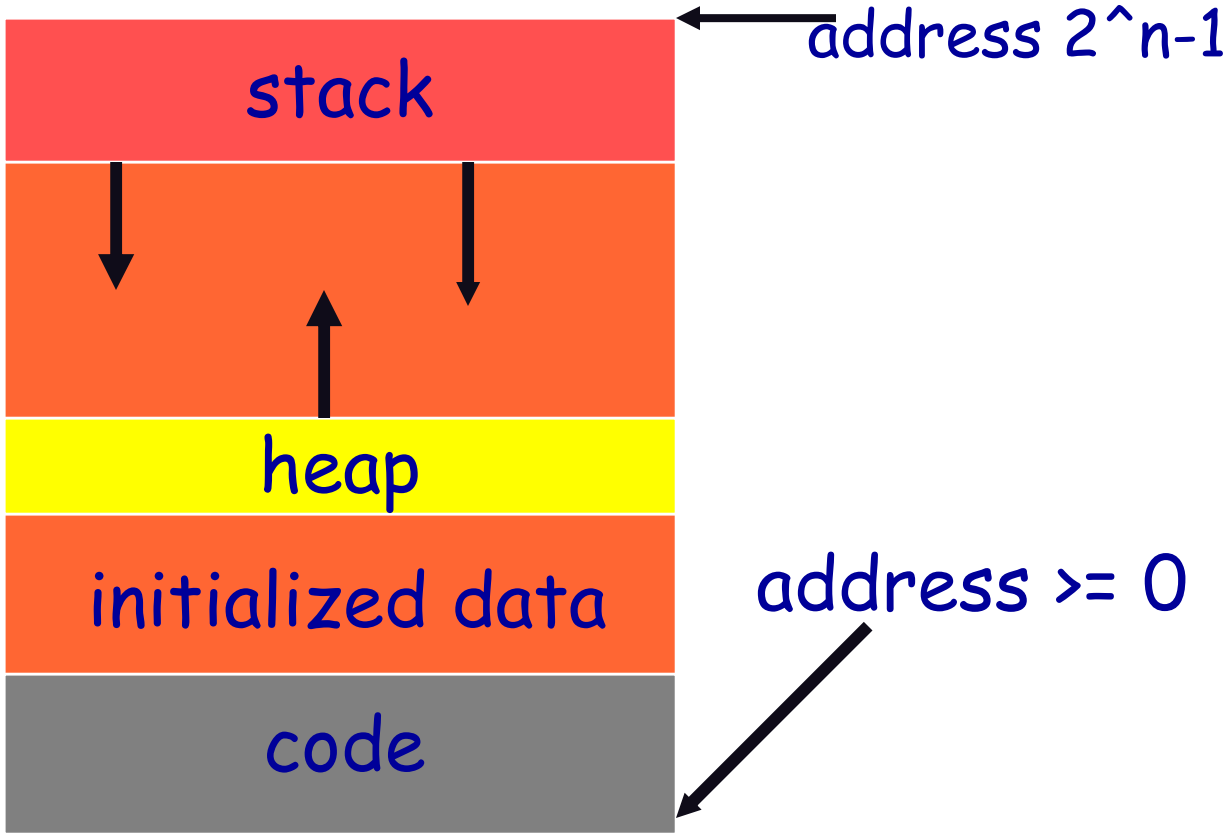
  Copies of same program running? Share code

  Multiple programs use same routines: share code (harder)

What does a process look like? (Unix)

- Process address space divided into "segments"

  text (code), data, heap (dynamic data), and stack



Why? (1) different allocation patterns; (2) separate code/data

Who builds what?

- Heap: constructed and layout by allocator (malloc)
  - compiler, linker not involved other than saying where it can start
  - namespace constructed dynamically and managed by programmer (names stored in pointers, and organized using data structures)
- Stack: alloc dynamic (proc call), layout by compiler
  - names are relative off of stack pointer
  - managed by compiler (alloc on proc entry, dealloc on exit)
  - linker not involved because name space entirely local: compiler has enough information to build it.
- Global data/code: allocation static (compiler), layout (linker)
  - compiler emits them and can form symbolic references between them ("jalr _printf")
  - linker lays them out, and translates references

Linkers (Linkage editors)

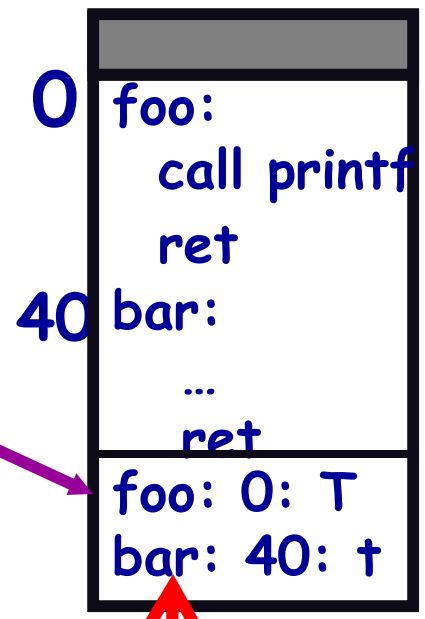
- Unix: ld
  - **usually hidden behind compiler**
- Three functions:
  - **collect together all pieces of a program**
  - **coalesce like segments**
  - **fix addresses of code and data so the program can run**
- Result: runnable program stored in new object file
- Why can't compiler do this?
  - **Limited world view: one file, rather than all files**
- Note *usually*:linkers only shuffle segments, but do not rearrange their internals.
  - **E.g., instructions not reordered; routines that are never called are not removed from a.out**

Simple linker: two passes needed

- Pass 1:
  - **coalesce like segments; arrange in non-overlapping mem.**
  - **read file's symbol table, construct global symbol table with entry for every symbol used or defined**
  - **at end: virtual address for each segment known (compute: start+offset)**
- Pass 2:
  - **patch refs using file and global symbol table**
  - **emit result**
- Symbol table: information about program kept while linker running
  - **segments: name, size, old location, new location**
  - **symbols: name, input segment, offset within segment**

Prob 1: where to put emitted objects? (def)

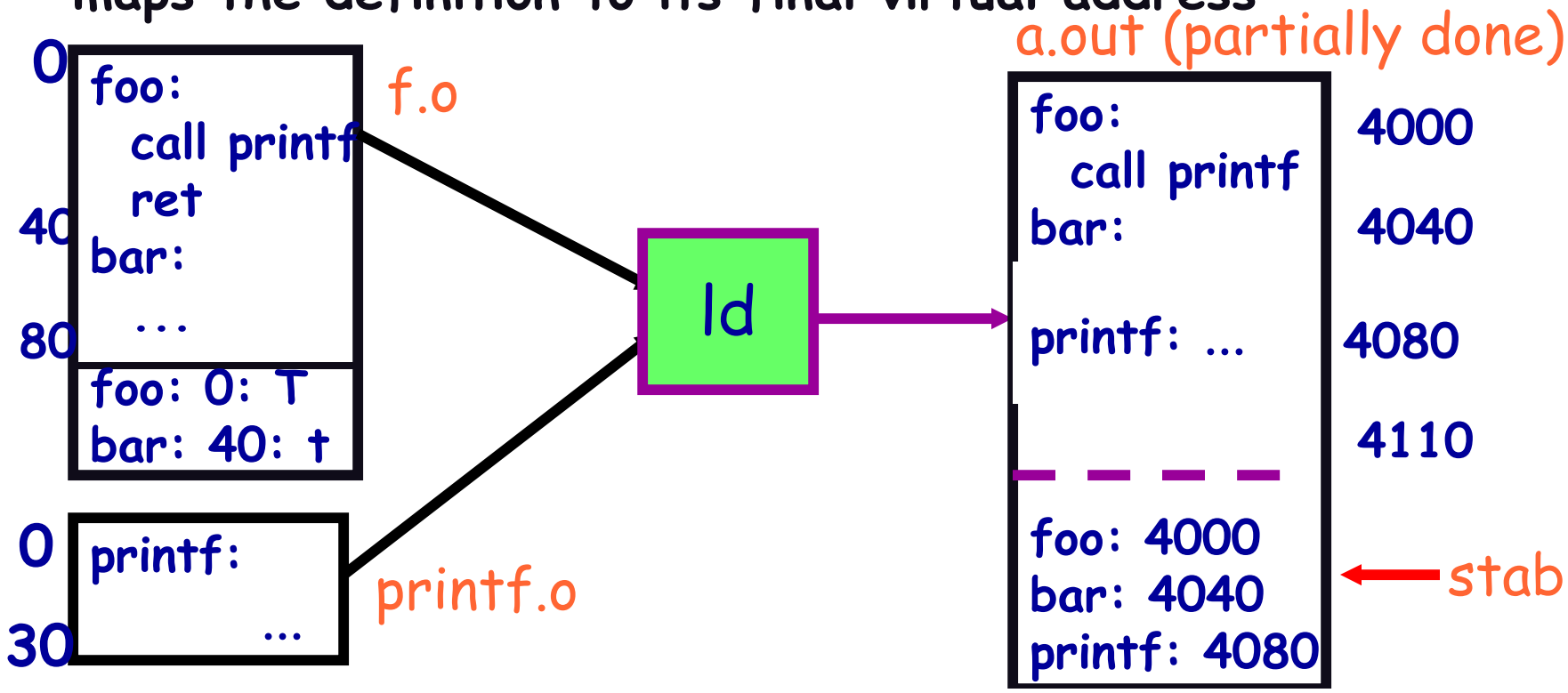
- Compiler:
  - doesn't know where data/code should be placed in the process's address space
  - assumes everything starts at zero
  - emits symbol table that holds the name and offset of each created object
  - routine/variables exported by the file are recorded global definition



- Simpler perspective:
  - code is in a big char array
  - data is in another big char array
  - compiler creates (object name, index) tuple for each interesting thing
  - linker then merges all of these arrays

Linker: where to put emitted objects?

- At link time, linker
  - determines the size of each segment and the resulting address to place each object at
  - stores all global definitions in a global symbol table that maps the definition to its final virtual address



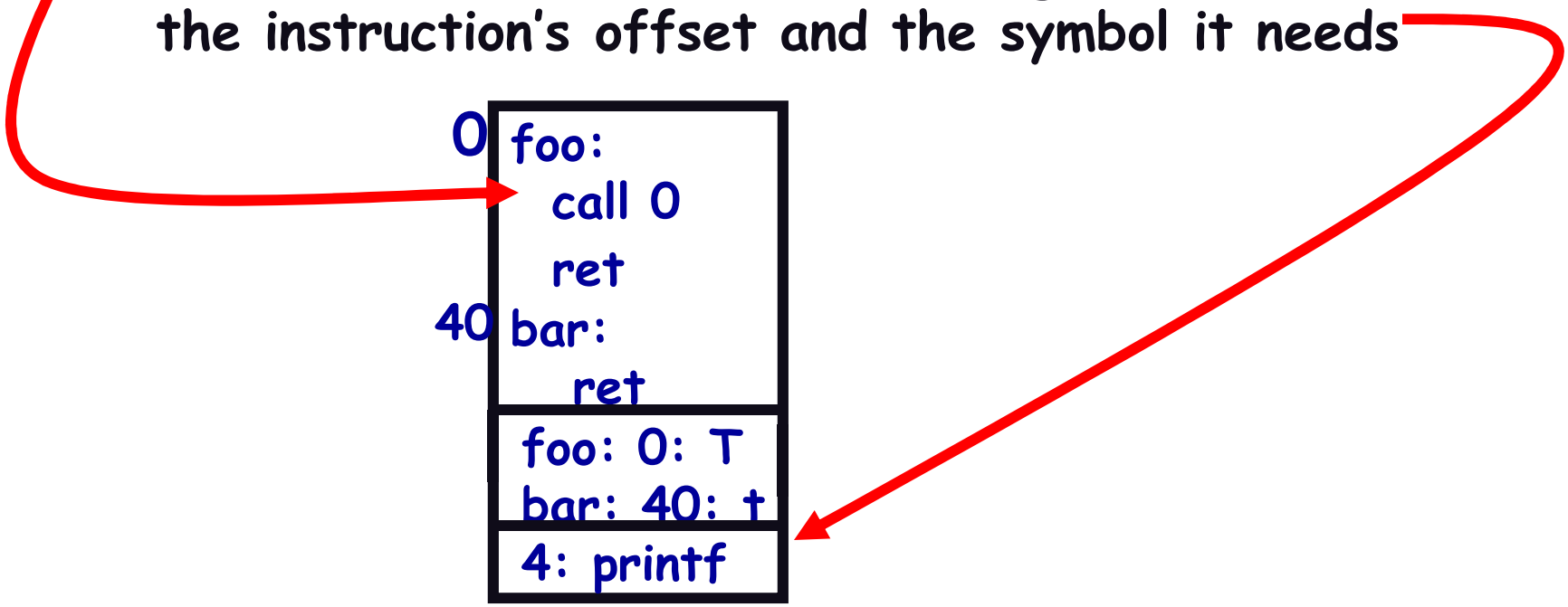
Problem 2: where is everything? (ref)

- How to call procedures or reference variables?

  E.g., call to printf needs a target addr

  compiler places a 0 for the address

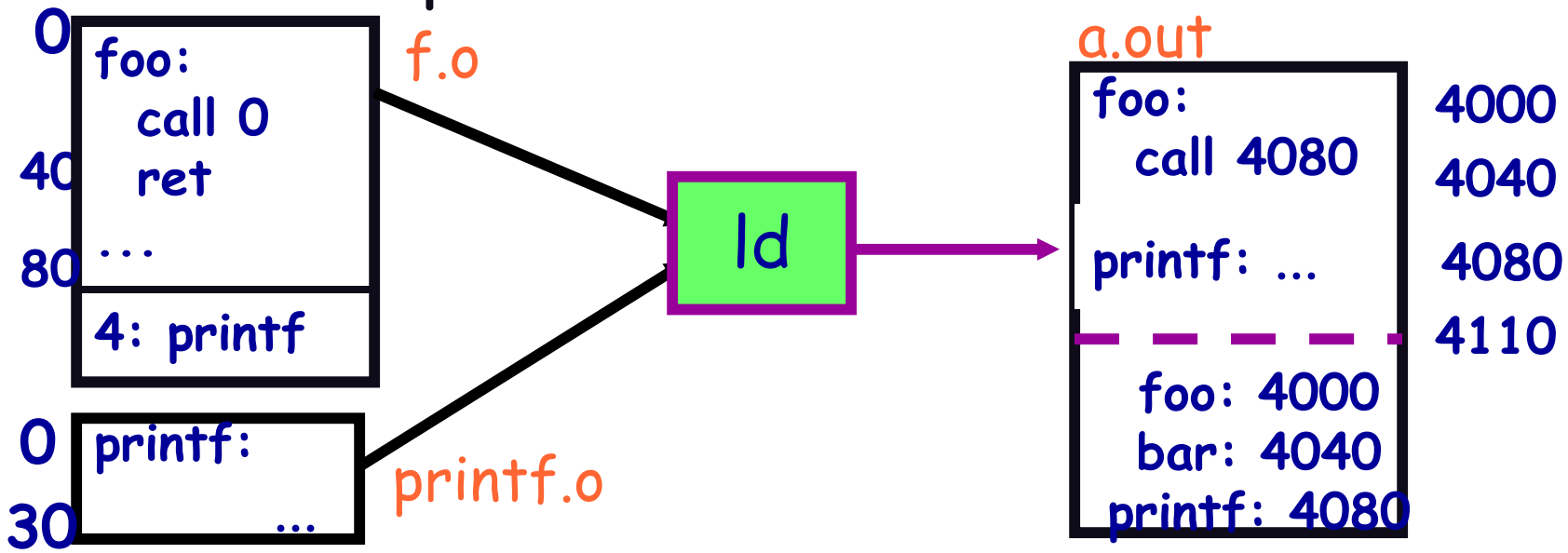
  emits an **external reference** telling the linker the instruction's offset and the symbol it needs



- At link time the linker patches every reference

Linker: where is everything?

- At link time the linker
  - records all references in the global symbol table
  - after reading all files, each symbol should have exactly one definition and 0 or more uses
  - the linker then enumerates all references and fixes them by inserting their symbol's virtual address into the reference's specified instruction or data location



Linking example: two modules and C lib

```
main.c:
   extern float sin();
   extern int printf(), scanf();
   float val;
   main() {
      static float x;
      printf("enter number");
      scanf("%f", &x);
      val = sin(x);
      printf("Sine is %f", val);
   }
```

```
math.c:
   float sin(float x) {
         float tmp1, tmp2;
         static float res;
         static float lastx;
         if(x != lastx) {
            lastx = x;
            … compute sin(x)…
         }
         return res;
   }
```

```
C library:
   int scanf(char *fmt, …) { … }
   int printf(char *fmt, …) { … }
```

Initial object files

Main.o:

```
   def: val @ 0:D        symbols
   def: main @ 0:T
   def: x @ 4:d
                         relocation
   ref: printf @ 8:T,12:T
   ref: scanf @ 4:T
   ref: x @ 4:T, 8:T
   ref: sin @ ?:T
   ref: val @ ?:T, ?:T
0  x:
4  val:                  data

0  call printf
4  call scanf(&x)        text
8  val = call sin(x)
12 call printf(val)
```

Math.o:

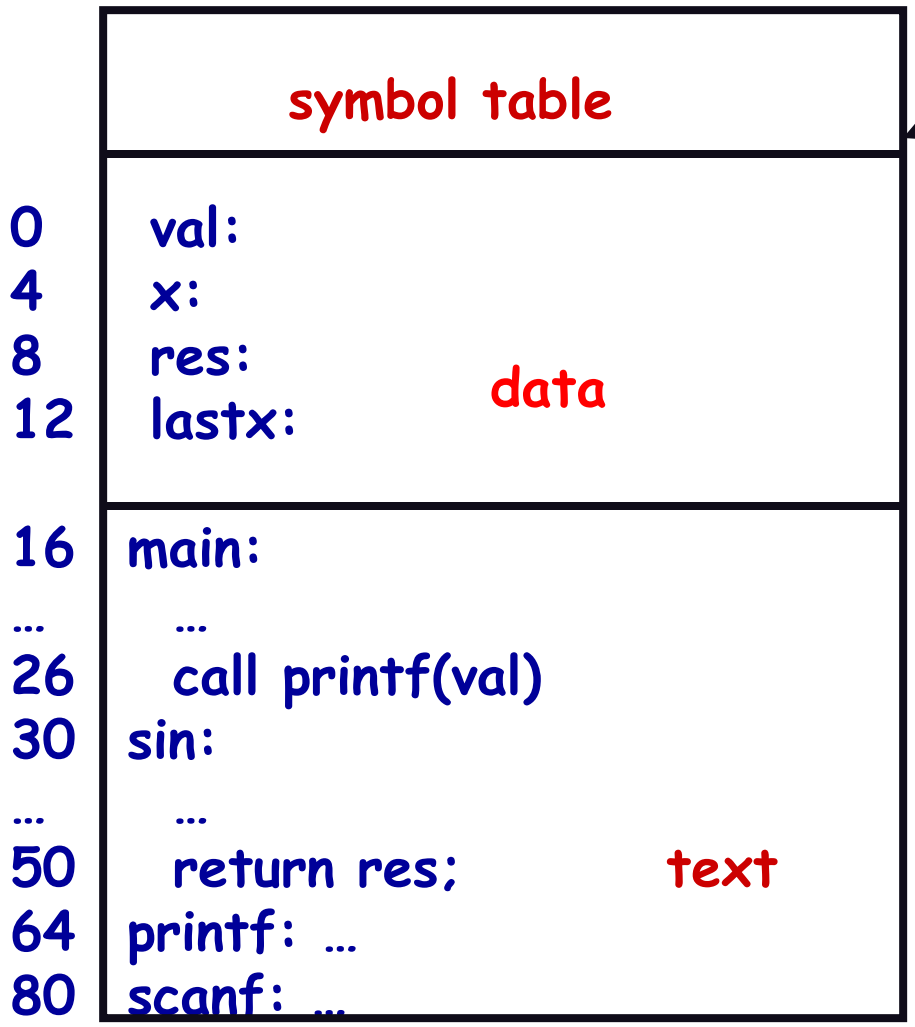
```
                         symbols
   def: sin @0:T
   def: res @ 0:d
   def: lastx @4:d
                         relocation
   ref: lastx@0:T,4:T
   ref res @24:T

0  res:                  data
4  lastx:

0   if(x != lastx)       text
4          lastx = x;
...        ... compute sin(x)...
24  return res;
```

Pass 1: Linker reorganization

a.out:



Starting virtual addr: 4000

Symbol table:

data starts @ 0
text starts @ 16
def: val @ 0
def: x @ 4
def: res @ 8
def: main @ 16
…
ref: printf @ 26
ref: res @ 50
…

(what are some other refs?)

Pass 2: relocation (insert virtual addresses)

"final" a.out:

Starting virtual addr: 4000

	symbol table	
0	val:	4000
4	x:	4004
8	res:	4008
12	lastx: data	4012
16	main: call 4064(4000)	4016
…	…	…
26	call ??(??) //printf(val)	4026
30	sin: text	4030
…	…	…
50	return load ??; // res	4050
64	printf: …	4064
80	scanf: …	4080

Symbol table:

data starts 4000
text starts 4016
def: val @ 0
def: x @ 4
def: res @ 8
def: main @ 14
def: sin @ 30
def: printf @ 64
def: scanf @80
…

(usually don't keep refs, since won't relink. Defs are for debugger: can be stripped out)

What gets written out

a.out:

		symbol table
16	main:	…
…		…
26		call 4064(4000)
30	sin:	…
…		…
50		return load 4008;
64	printf:	…
80	scanf:	…

virtual addr: 4016

4016	
…	
4026	
4030	
…	
4050	
4064	
4080	

Symbol table:

initialized data = 4000
uninitialized data = 4000
text = 4016
def: val @ 0
def: x @ 4
def: res @ 8
def: main @ 14
def: sin @ 30
def: printf @ 64
def: scanf @80

Uninitialized data allocated and zero filled at load time.

Types of relocation

- Place final address of symbol here
  - **data example: extern int y, *x = &y;**
    - **y gets allocated an offset in the uninitialized data segment**
    - **x is allocated a space in the initialized data seg (I.e., space in the actual executable file). The contents of this space are set to y's computed virtual address.**
  - **code example: call foo becomes call 0x44**
    - **the computed virtual address of foo is stuffed in the binary encoding of "call"**
- Add address of symbol to contents of this location
  - **used for record/struct offsets**
  - **example: q.head = 1 to move #1, q+4 to move #1, 0x54**
- add diff between final and original seg to this location
  - **segment was moved, "static" variables need to be reloc'ed**

Linking variation 0: dynamic linking

- Link time isn't special, can link at runtime too
  - **Get code not available when program compiled**
  - **Defer loading code until needed**

```
void foo(void) { puts("hello"); }
```

→ gcc -c foo.c →

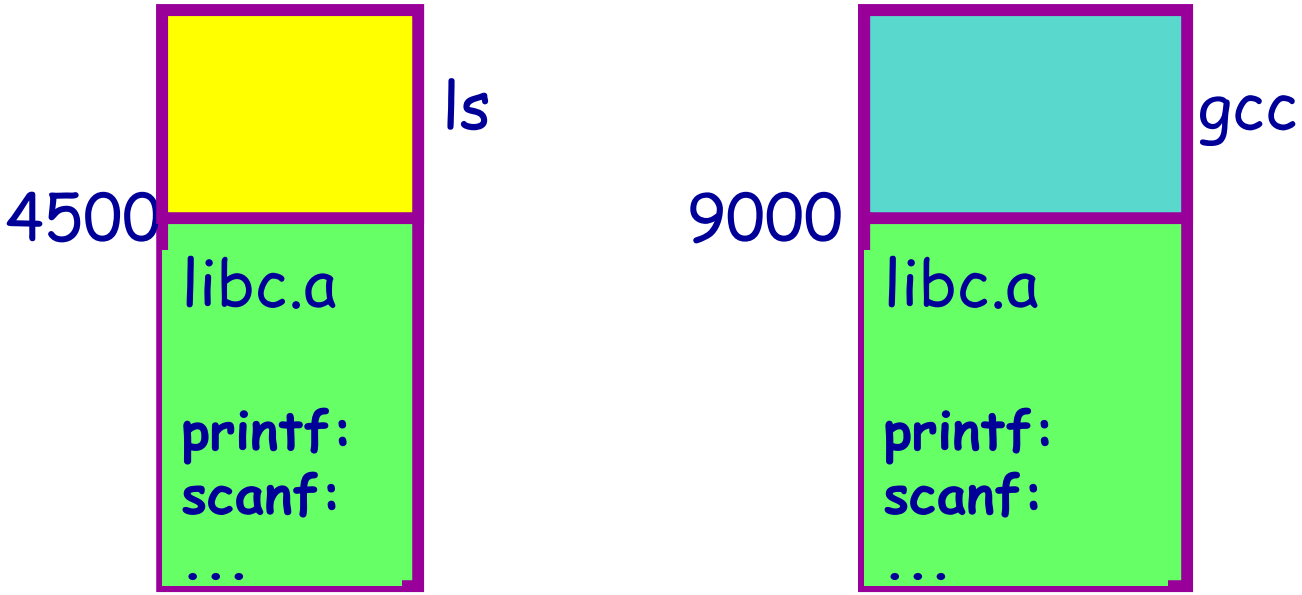
```
foo:
  call puts
```

```
void *p = dlopen ("foo.o", RTLD_LAZY);
void (*fp)(void) = dlsym(p, "foo");
fp();
```

Issues: what happens if can't resolve? How can behavior differ compared to static linking? Where to get unresolved syms (e.g., "puts") from?

Linking variation 1: static shared libraries

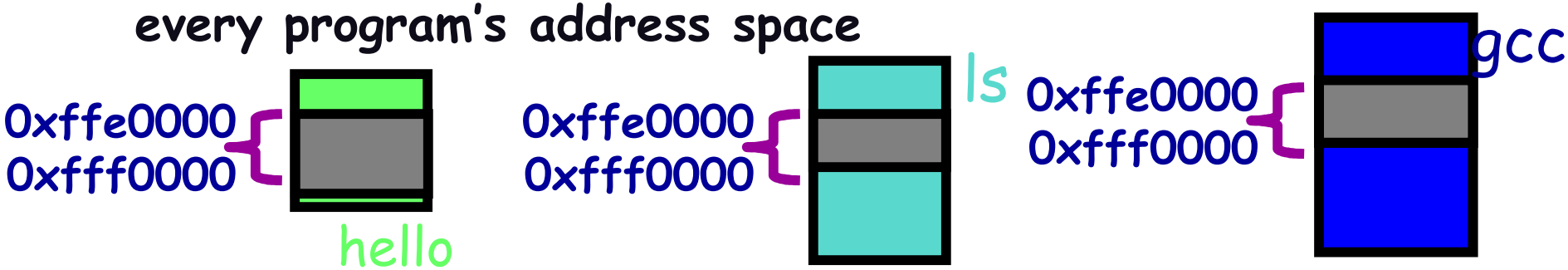
- Observation: everyone links in standard libraries (libc.a.), these libs consume space in every executable.



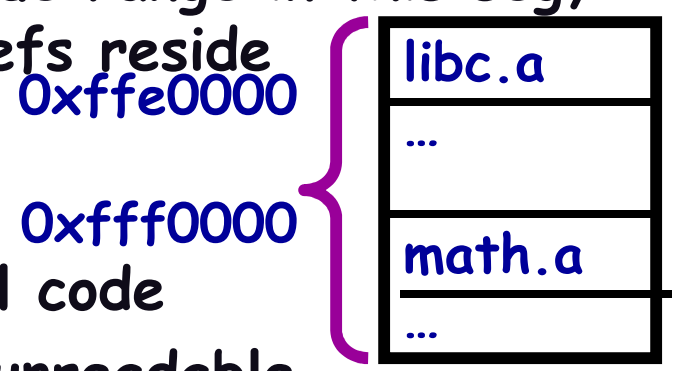
- Insight: we can have a single copy on disk if we don't actually include lib code in executable

Static shared libraries

Define a "shared library segment" at same address in every program's address space



Every shared lib is allocated a unique range in this seg, and computes where its external defs reside



Linker links program against lib (why?) but does not bring in actual code

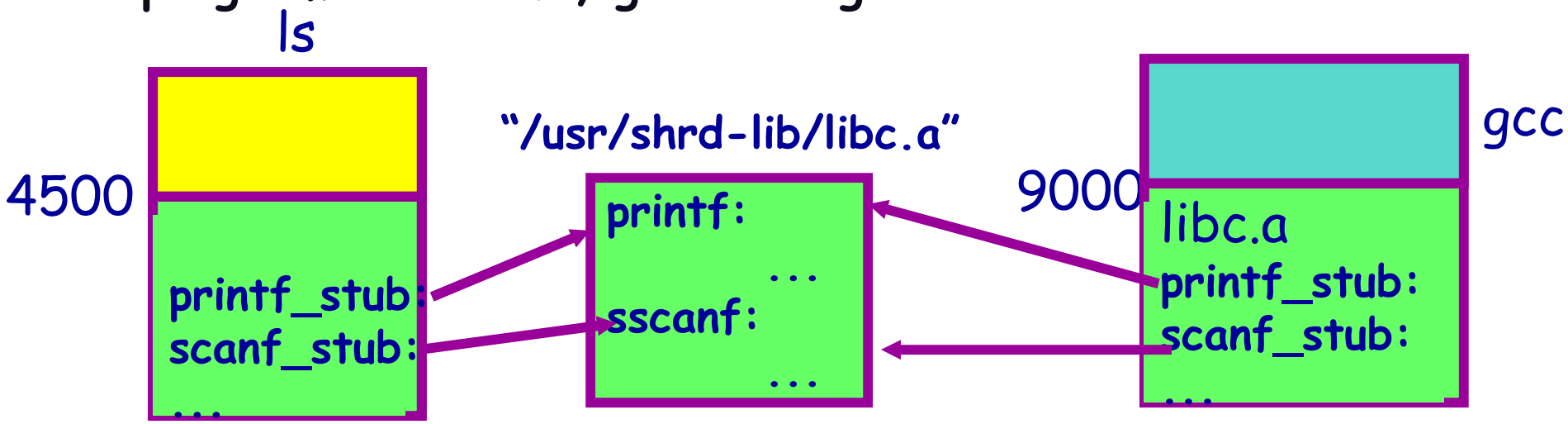
Loader marks shared lib region as unreadable

When process calls lib code, seg faults: enclosed linker brings in lib code from known place & maps it in.

Now different running programs can now share code!

Linking variation 2: dynamic shared libs

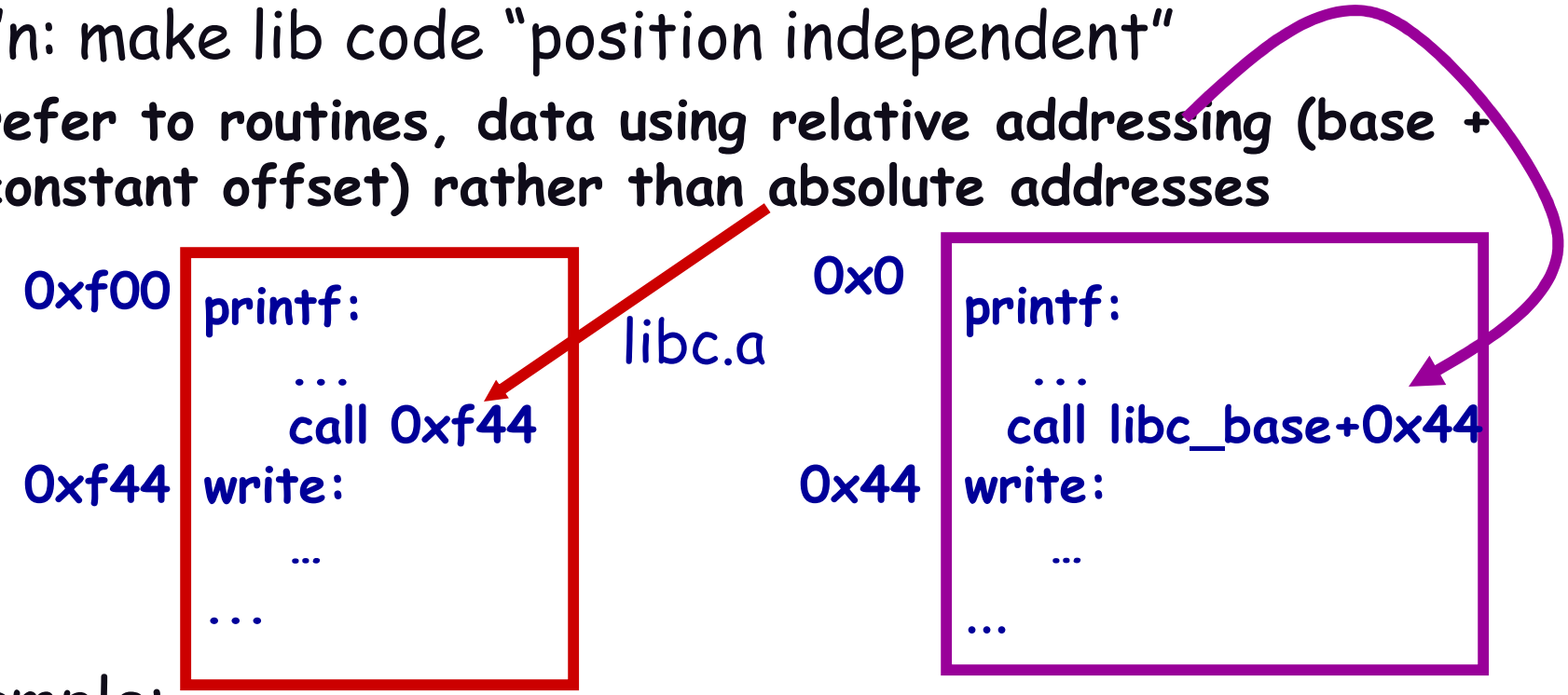
- Problem: static shared libraries require system-wide pre-allocation address space: clumsy
  - **We want to link code anywhere in address space**
- Problem 1: linker won't know how to resolve refs
  - **do resolution at runtime**
  - **link in stubs that know where to get code from**
  - **program calls stub, goes and gets code**



Problem 2: Dynamic shared libraries

- Code must simultaneously run at different locations!
- Sol'n: make lib code "position independent"

  refer to routines, data using relative addressing (base + constant offset) rather than absolute addresses



- Example:

  internal call "call 0xf44" become "call lib_base + 0x44"

  "lib_base" contains the base address of library (private to each process) & 0x44 is called routine's internal offset

Code = data, data = code

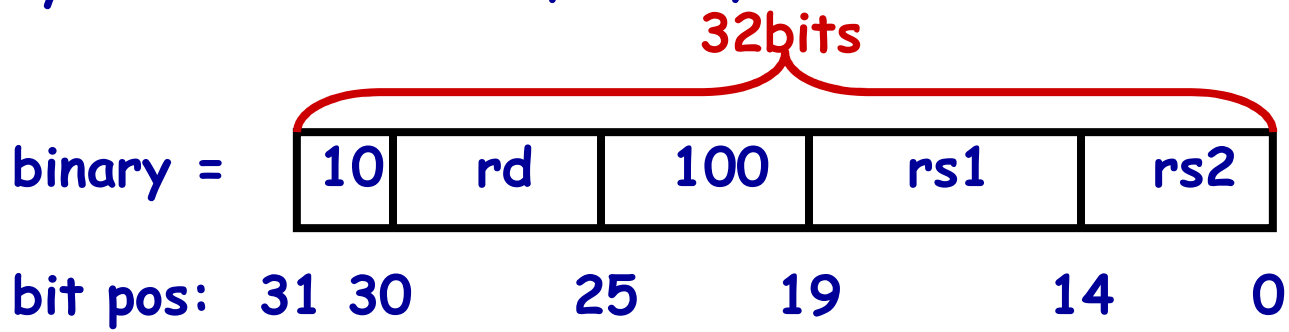
- No inherent difference between code and data
  - **Code is just something that can be run through a CPU without causing an "illegal instruction fault"**
  - **Can be written/read at runtime just like data "dynamically generated code"**
- Why? Speed (usually)
  - **Big use: eliminate interpretation overhead. Gives 10-100x performance improvement**
  - **Example: Just-in-time compilers for java.**
  - **In general: optimizations thrive on information. More information at runtime.**
- The big tradeoff:
  - **Total runtime = code gen cost + cost of running code**

How?

- Determine binary encoding of desired assembly inst's

  SPARC: sub instruction

  symbolic = "sub rdst, rsrc1, rsrc2"



- Write these integer values into a memory buffer

```
unsigned code[1024], *cp = &code[0];
/* sub %g5, %g4, %g3 */
*cp++ = (2<<30) | (5<<25) | (4<<19) |(4<<14) | 3;
...
```

- Jump to the address of the buffer!

```
((int (*)())code)();  /* cast to function pointer and call. */
```

Linking Summary

- Compiler: generates 1 object file for each source file
  - **Problem: incomplete world view**
  - **Where to put variables and code? How to refer to them?**
  - **Names definitions symbolically ("printf"), refers to routines/variable by symbolic name**
- Linker: combines all object files into 1 executable file
  - **big lever: global view of everything. Decides where everything lives, finds all references and updates them**
  - **Important interface with OS: what is code, what is data, where is start point?**
- OS loader reads object files into memory:
  - **allows optimizations across trust boundaries (share code)**
  - **provides interface for process to allocate memory (sbrk)**