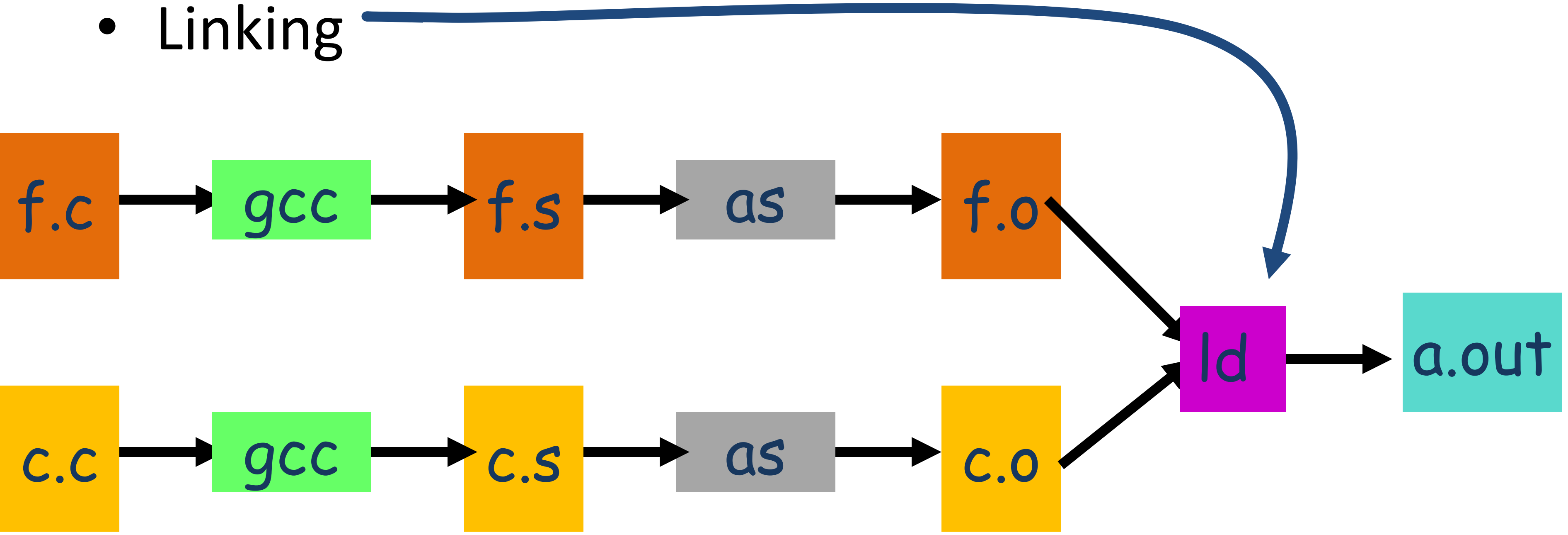
CSL373: Operating Systems Linking

Today

- Linking



e.g., f.o = hello.o (uses printf); c.o = libc.o (implements printf)

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 ; man elf
  - run "nm" on a few .o and a.out files

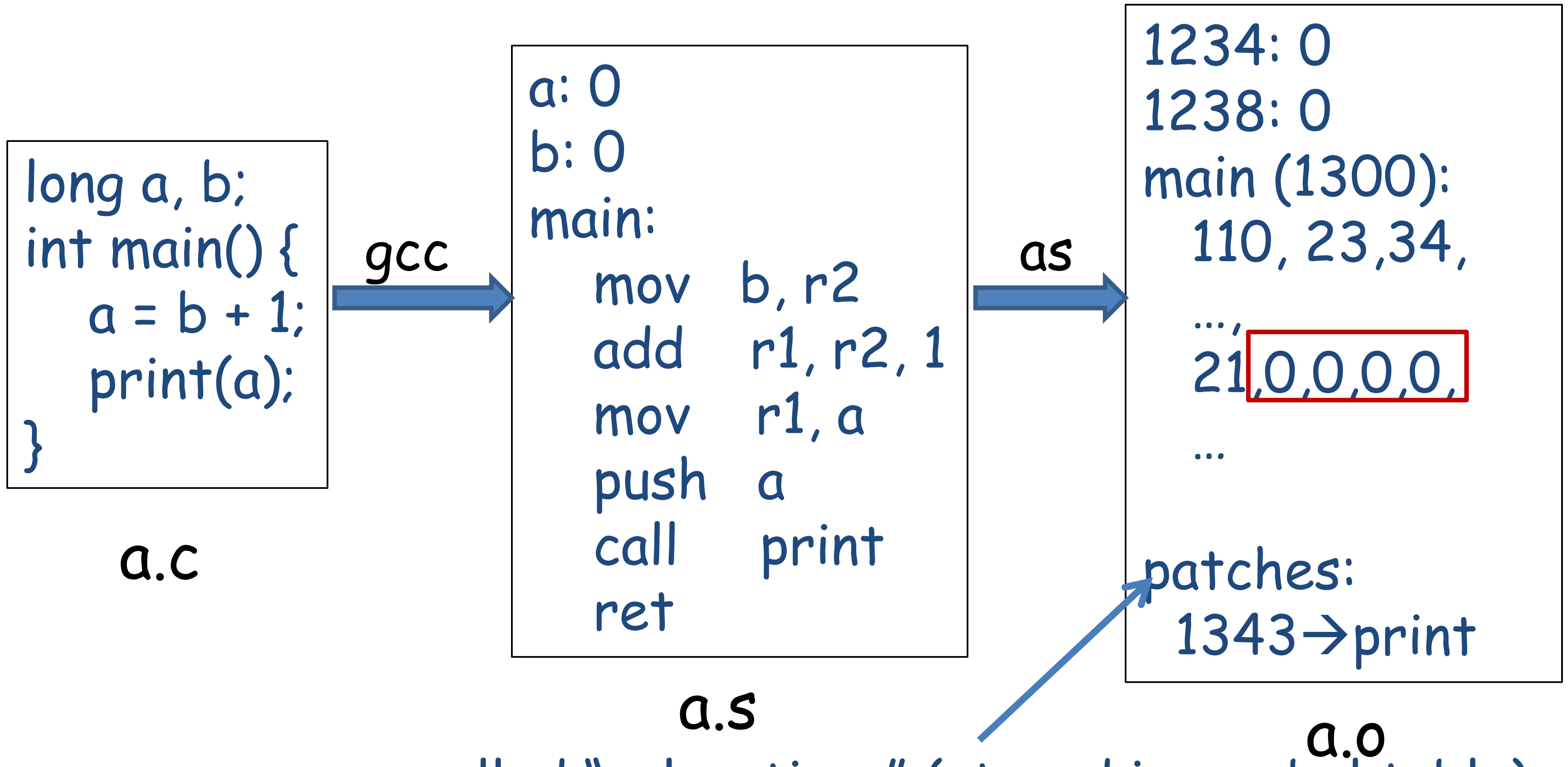
Linking as our first naming system

- 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, …
  - www.cse.iitd.ernet.in resolved to 10.20.33 using DNS
  - Your name resolved to grade (value) using spreadsheet

Programming language view



```
long a, b;
int main() {
    a = b + 1;
    print(a);
}
```

```
a: 0
b: 0
main:
    mov   b, r2
    add   r1, r2, 1
    mov   r1, a
    push  a
    call  print
    ret
```

```
1234: 0
1238: 0
main (1300):
   110, 23,34,
   ...,
   21,0,0,0,0,
   ...

patches:
  1343→print
```

called "relocations" (stored in symbol table)

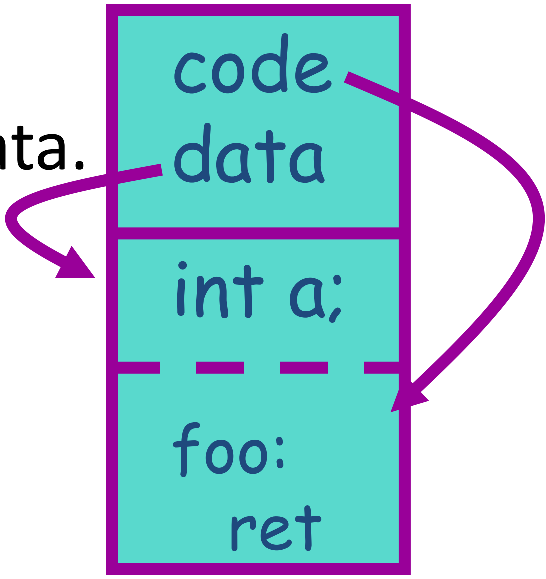
Perspectives on information in memory

- Programming language view:
  - instructions: specify operations to perform
  - variables: operands that can change over time
  - constants: operands that never change
- Address 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?
  - Compile time
  - Link time
  - Run time

early to late

How is a process specified?

- shell$ ./a.out
  - A process is created from an executable
- The executable file (a.out) is the interface between the linker and the OS
  - specifies, where is code. where is data.
  - where should they live?

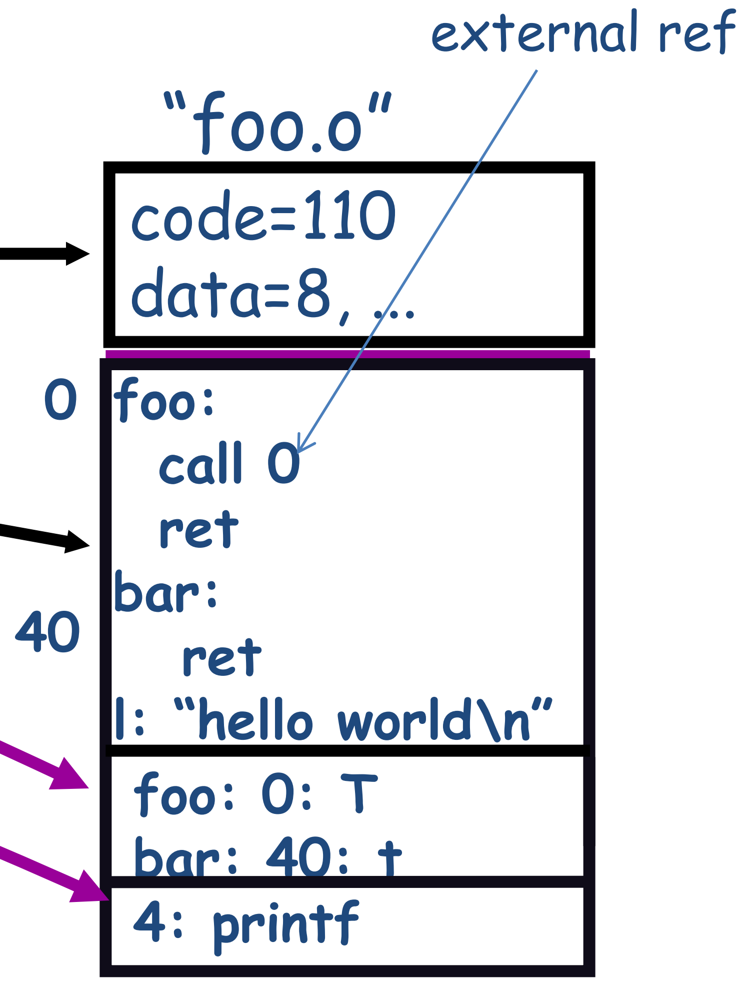


Linker: object files → executable

Header: code/data size, symbol table offset

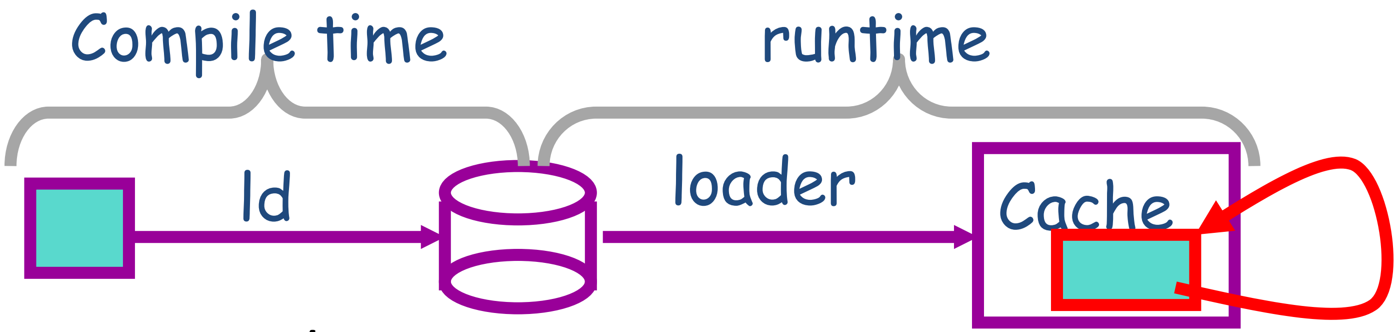
Object code: instructions and data gen'd by compiler

Symbol table:
external defs (exported objects in file)
external refs (global symbols 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

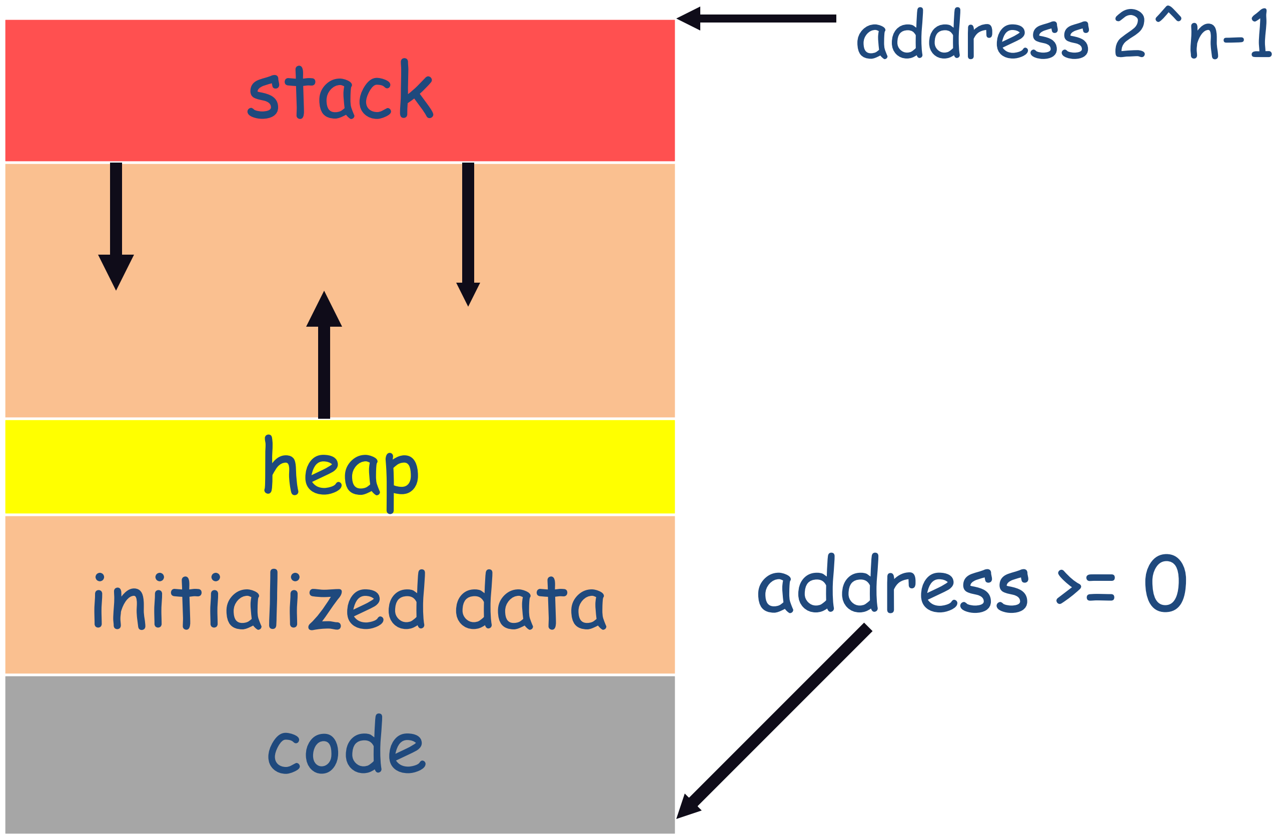
Copies of same program running? Share code

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 as data structures)

  OS provides sbrk() system call to allocate a new chunk of memory for the heap (called internally by malloc()).

Who builds what?

- Stack: allocated dynamically (proc call), layout by compiler

  names are relative off 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.

e.g.,

```
void foo(void) {
    long a, b;
     a = a + 1;
     b = b * 2;
}
```

$\longrightarrow$

```
foo:
     add   8, sp
     add   1, [sp+4]
     mul    2, [sp]
     …
```

sp = stack pointer

Local variable 'a' stored at [sp+4], local variable 'b' stored at [sp]

Who builds what?

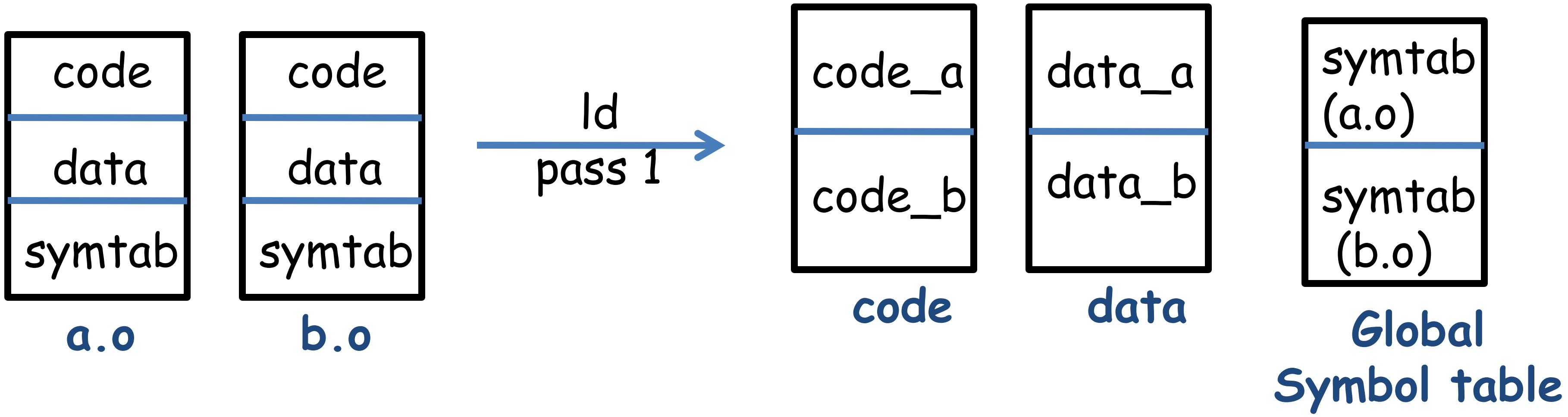
- Global data/code: allocation static (compiler), layout (linker)
  - Compiler emits them and can form symbolic references between them (“call printf”)
  - Linker lays them out, and translates references

Linkers (linkage editors)

- Unix: ld
  - Usually hidden behind compiler (try “gcc –v”)
- 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 compiler can’t 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)



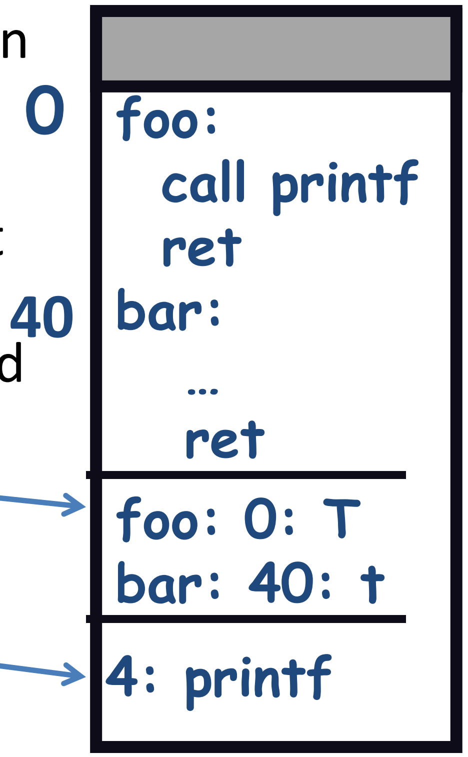
Simple linker: pass 2

- Pass 2:
  - Patch refs using file and global symbol table
  - (In the object files, the symbol table contained only offsets inside the segments. Now the linker knows the full virtual address, once segment is relocated.)
  - 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

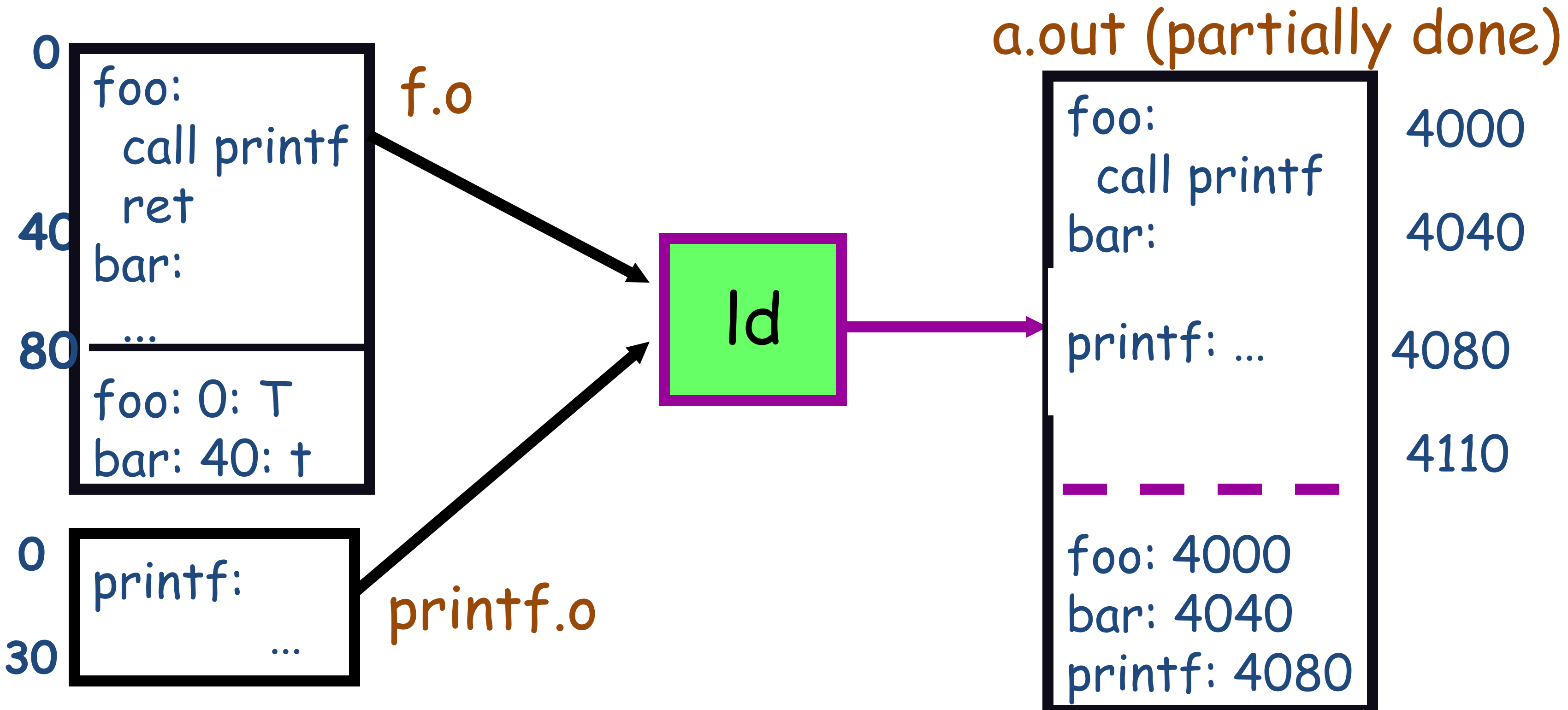
Why have a symbol table?

- 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 <u>exported</u> by the file are recorded as **global definition**
  - Routine/variables <u>used</u> by the file are recorded as **references**.
- 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



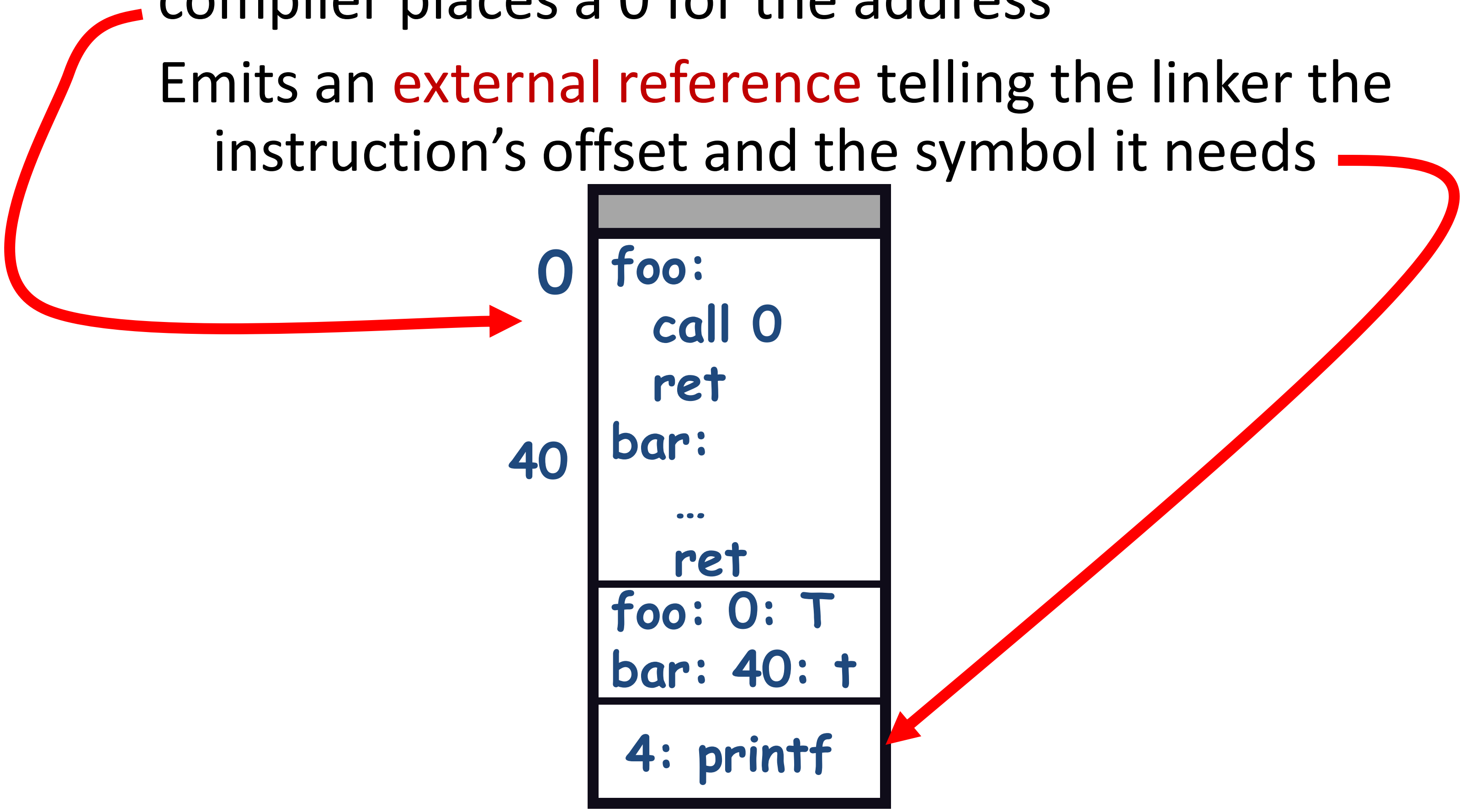
Problem 2: where is everything? (ref)

- How to call procedures or reference variables?

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

  compiler places a 0 for the address

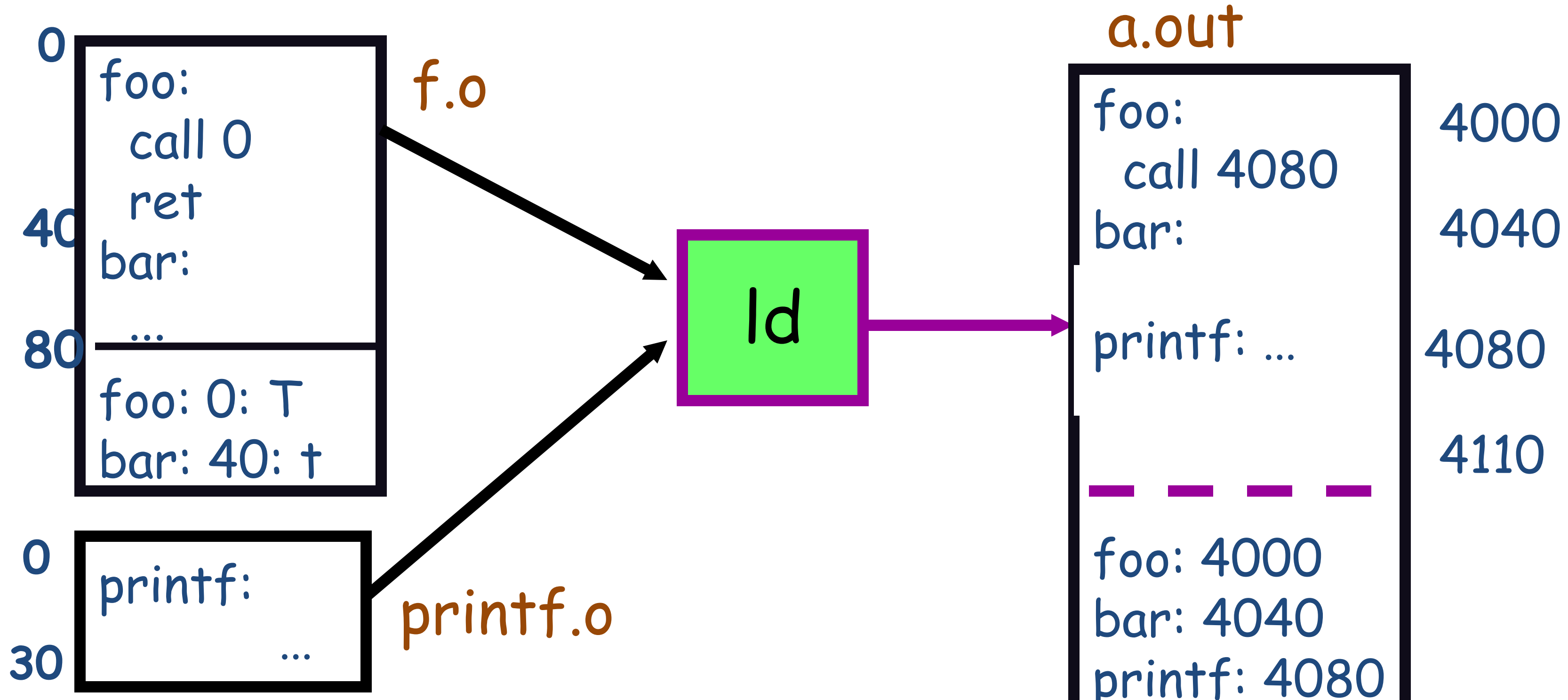
  Emits an external reference telling the linker the instruction's offset and the symbol it needs



- At link time, the linker patches everything

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



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);
    }
```

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

```
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;
    }
```

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)
```

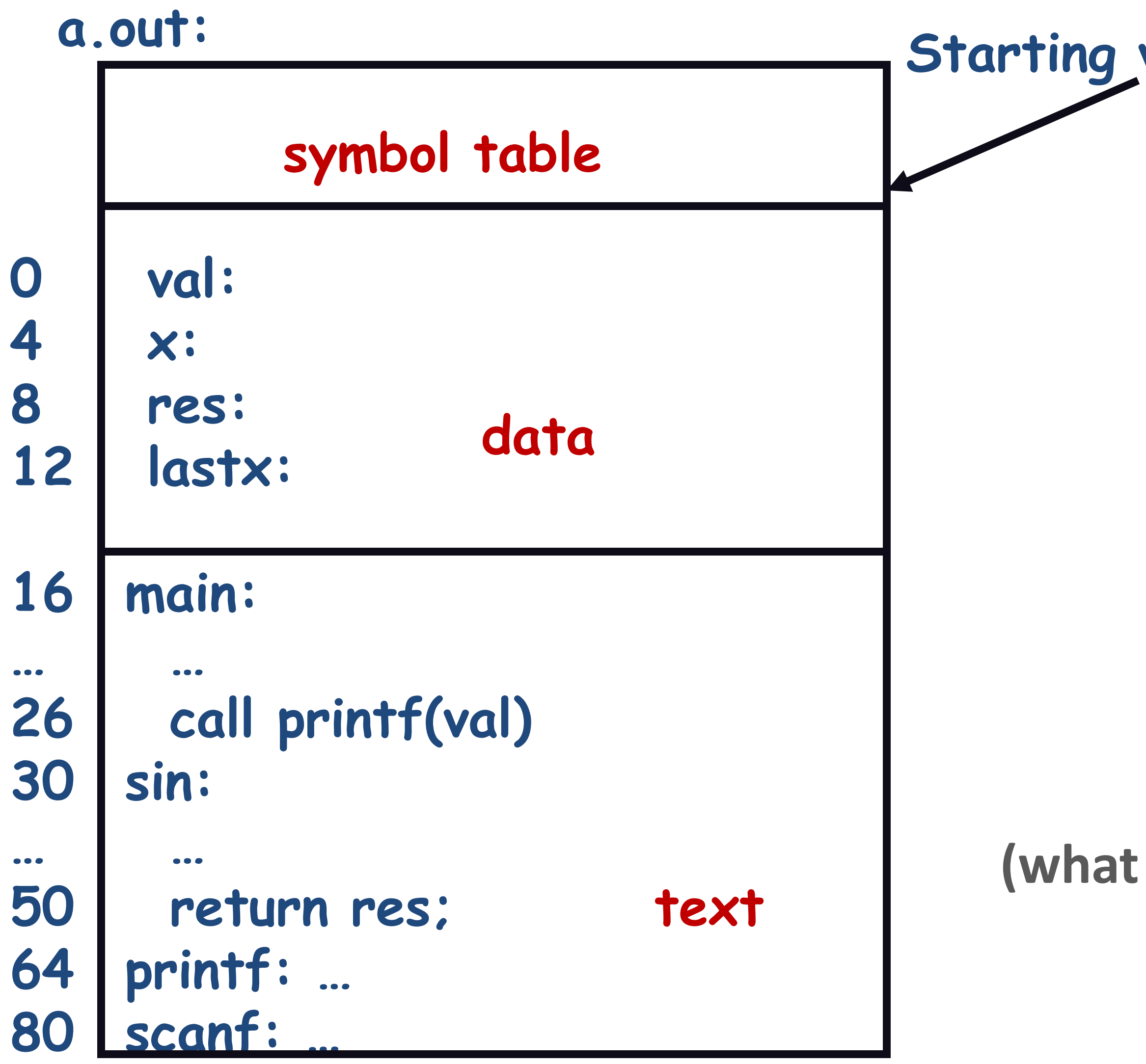
Math.o:

```
           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



Symbol table:

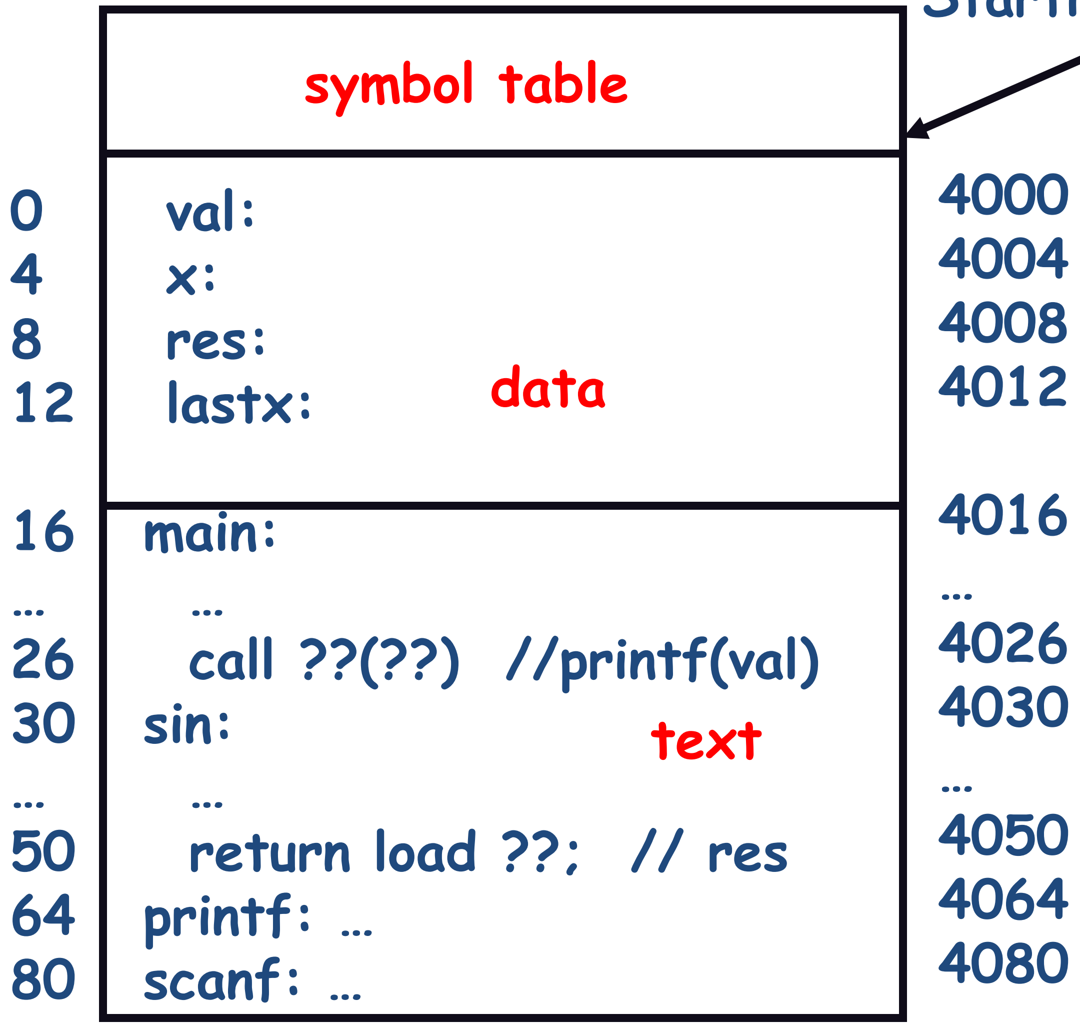
data starts @ 0
text starts @ 16
def: val @ 0
def: x @ 4
def: res @ 8
def: main @ 16
…
ref: printf @ 26
ref: res @ 50
…

(what are some other refs?)

Pass 2: relocation (insert virtual addrs)

"final" a.out:

Starting virtual addr: 4000

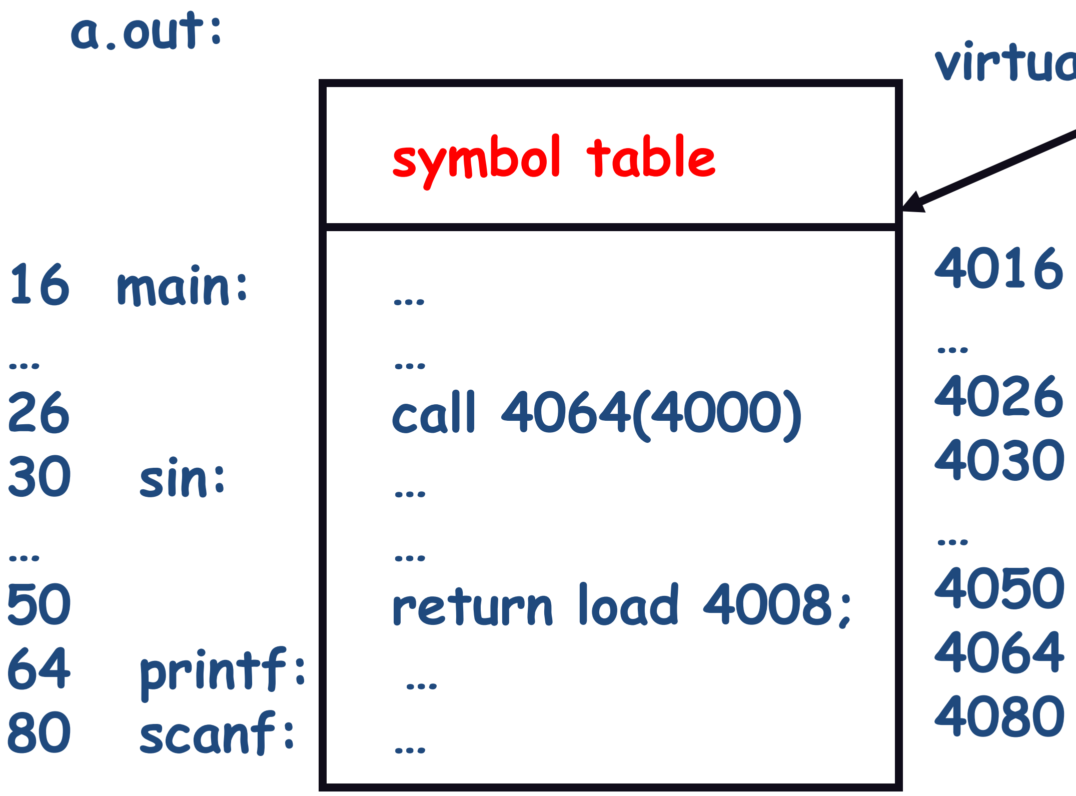


Symbol table:

- 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:
                                           virtual addr: 4016
            symbol table

16  main:     ...                      4016
...           ...                      ...
26            call 4064(4000)          4026
30   sin:     ...                      4030
...           ...                      ...
50            return load 4008;        4050
64   printf:   ...                     4064
80   scanf:   ...                      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 space of initialized data segment (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

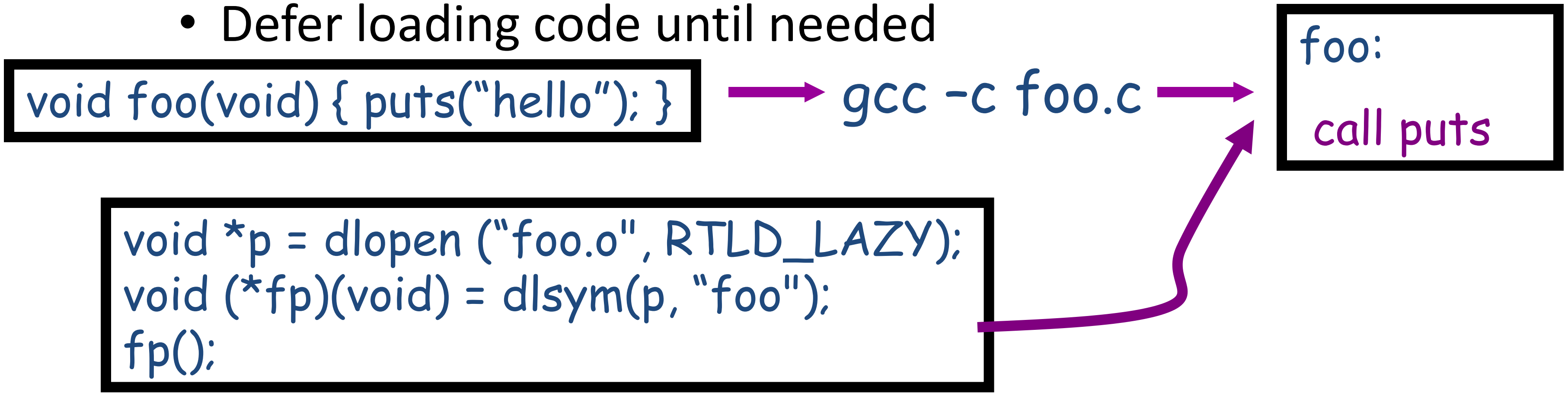
- Add diff between final and original seg to this location

  segment was moved, "static" variables need to be reloc'ed

```
sbansal@sri ~$ cat /proc/29052/maps                              [application=bash]
00110000-00111000 r-xp 00110000 00:00 0          [vdso]
00bcd000-00be6000 r-xp 00000000 fd:00 135235     /lib/ld-2.5.so
00be6000-00be7000 r-xp 00018000 fd:00 135235     /lib/ld-2.5.so
00be7000-00be8000 rwxp 00019000 fd:00 135235     /lib/ld-2.5.so
00bea000-00d21000 r-xp 00000000 fd:00 135236     /lib/libc-2.5.so
00d21000-00d23000 r-xp 00137000 fd:00 135236     /lib/libc-2.5.so
00d23000-00d24000 rwxp 00139000 fd:00 135236     /lib/libc-2.5.so
00d24000-00d27000 rwxp 00d24000 00:00 0
00d52000-00d54000 r-xp 00000000 fd:00 135237     /lib/libdl-2.5.so
00d54000-00d55000 r-xp 00001000 fd:00 135237     /lib/libdl-2.5.so
00d55000-00d56000 rwxp 00002000 fd:00 135237     /lib/libdl-2.5.so
05cb9000-05cbc000 r-xp 00000000 fd:00 135248     /lib/libtermcap.so.2.0.8
05cbc000-05cbd000 rwxp 00002000 fd:00 135248     /lib/libtermcap.so.2.0.8
08047000-080f2000 r-xp 00000000 fd:00 2709149    /bin/bash
080f2000-080f7000 rw-p 000ab000 fd:00 2709149    /bin/bash
080f7000-080fc000 rw-p 080f7000 00:00 0
0987f000-098bf000 rw-p 0987f000 00:00 0
b7d2f000-b7f2f000 r--p 00000000 fd:00 4997969    /usr/lib/locale/locale-archive
b7f2f000-b7f64000 r--s 00000000 fd:00 2906265    /var/db/nscd/passwd
b7f64000-b7f65000 rw-p b7f64000 00:00 0
b7f75000-b7f76000 rw-p b7f75000 00:00 0
b7f76000-b7f7d000 r--s 00000000 fd:00 32900      /usr/lib/gconv/gconv-modules.cache
b7f7d000-b7f7e000 rw-p b7f7d000 00:00 0
bf9b0000-bf9c5000 rw-p bf9b0000 00:00 0          [stack]
```

Linking variation 0: dynamic linking

- Link time isn't special, can link at runtime too
  - Why?
    - Get code not available when program compiled
    - Can use different library code for different environs
    - Defer loading code until needed



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

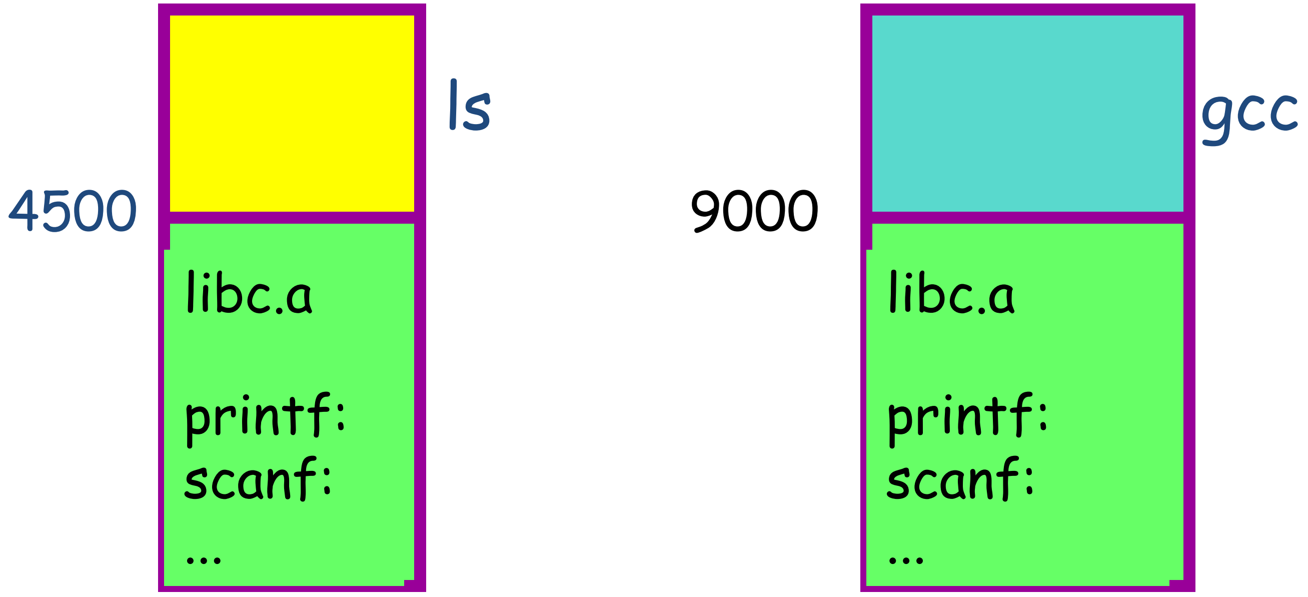
```
foo:
 call puts
```

```
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

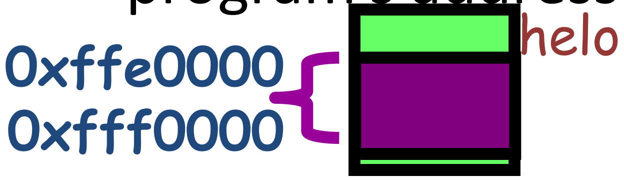
- Observation: everyone links in standard libraries (e.g., libc.a), these libs consume space in every executable.

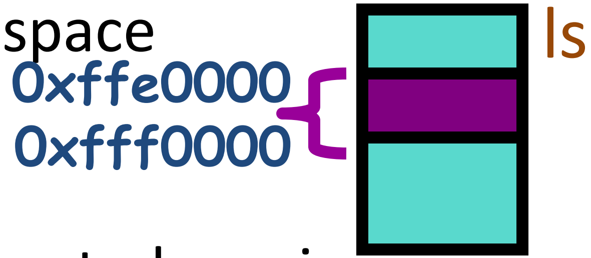


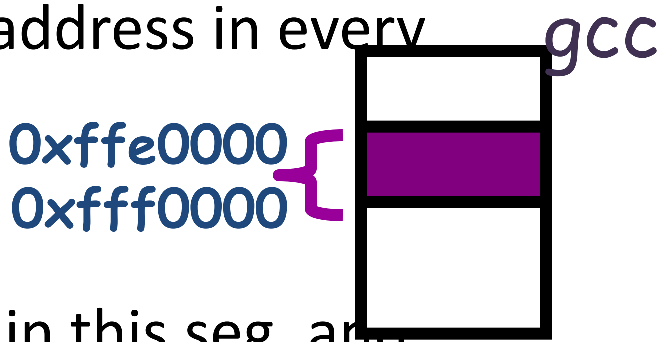
- Insight: we can have a single copy on disk if we don’t actually include lib code in executable

Static shared libraries

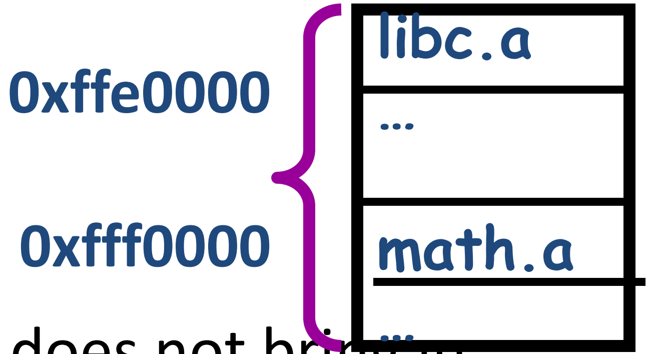
Define a "shared library segment" at same address in every program's address space







Every shared lib is allocated a unique range in this seg, and computes where 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.

So? Different running programs can now share code!

Linking variation 2: dynamic shared libs

- Problem: static shared libraries require system-wide pre-allocation of address space: clumsy

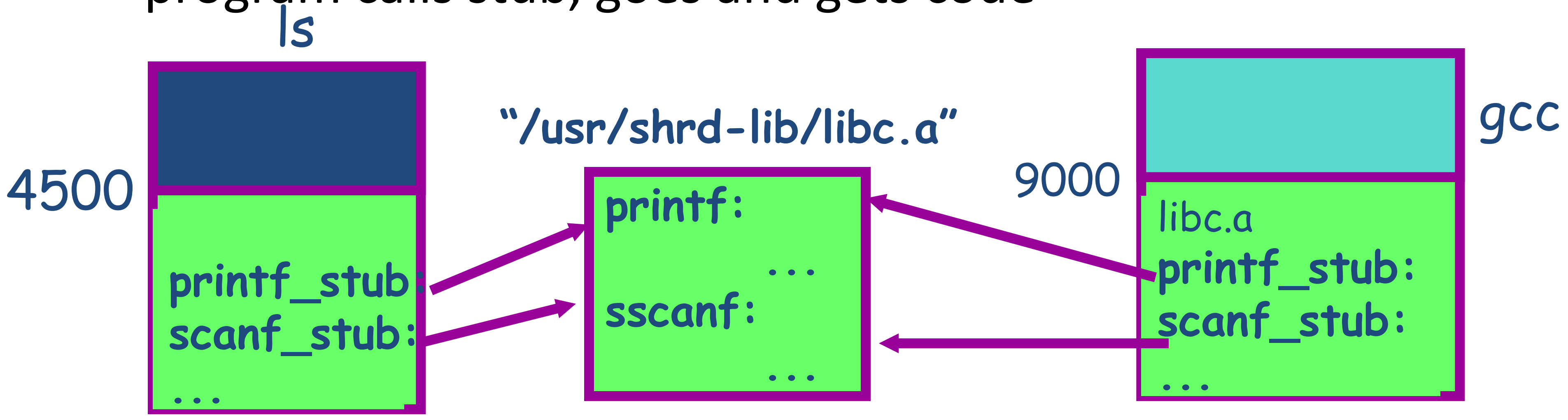
  We want to link code anywhere in address space

- Problem 1: linker won't know how to resolve refs

  do resolution at runtime

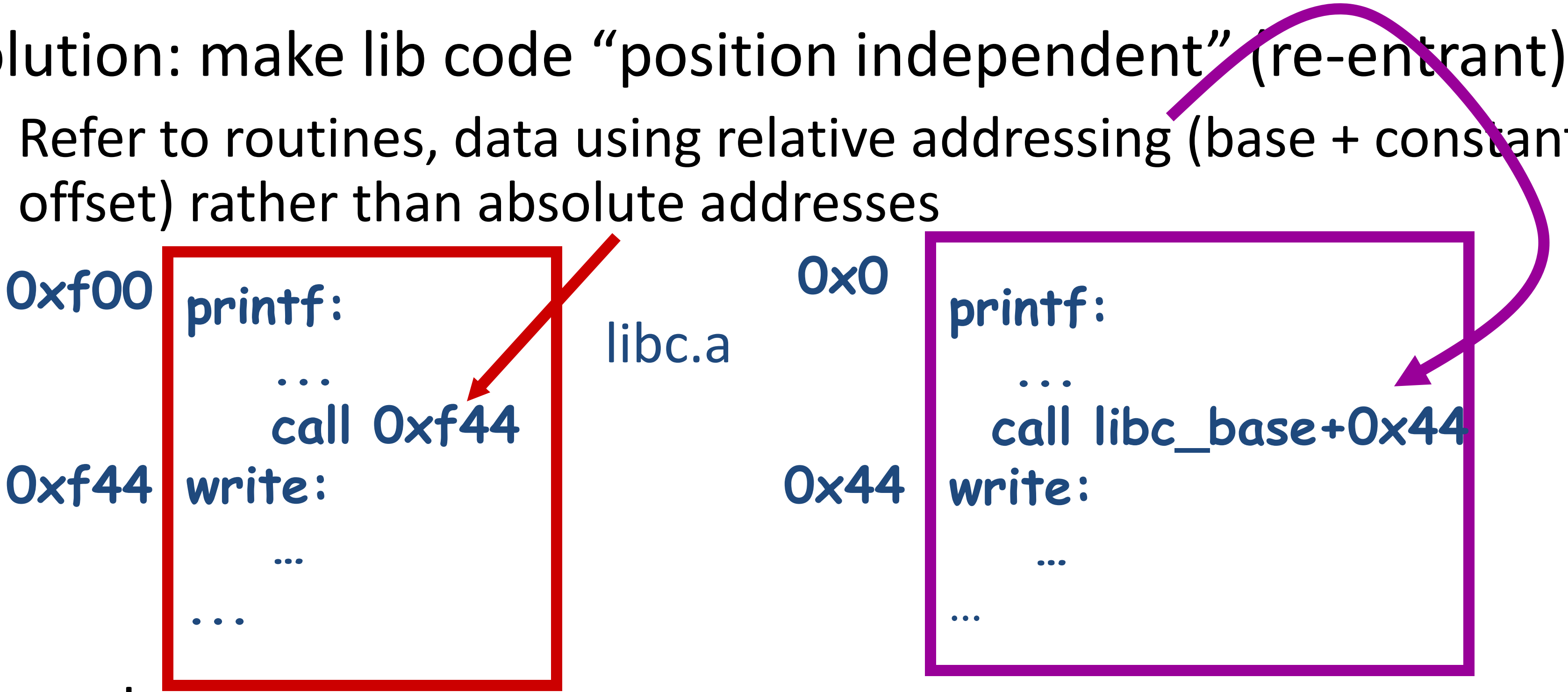
  link in stubs that know where to get code from

  program calls stub, goes and gets code



Problem 2: Dynamic shared libraries

- Code must simultaneously run at different locations!
- Solution: make lib code "position independent" (re-entrant)
  - Refer to routines, data using relative addressing (base + constant offset) rather than absolute addresses



- Example:
  - Internal call "call 0xf44" becomes "call lib_base + 0x44"
  - "lib_base" contains the base address of library (private to each process) and 0x44 is the 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 dynamically generated code? Speed (usually)
  - Big use: eliminate interpretation overhead. Gives 10-100x performance improvement
  - Example: Just-in-time compilers for Java
  - In general: better information ➔ better optimization. More information at runtime

- The big tradeoff:

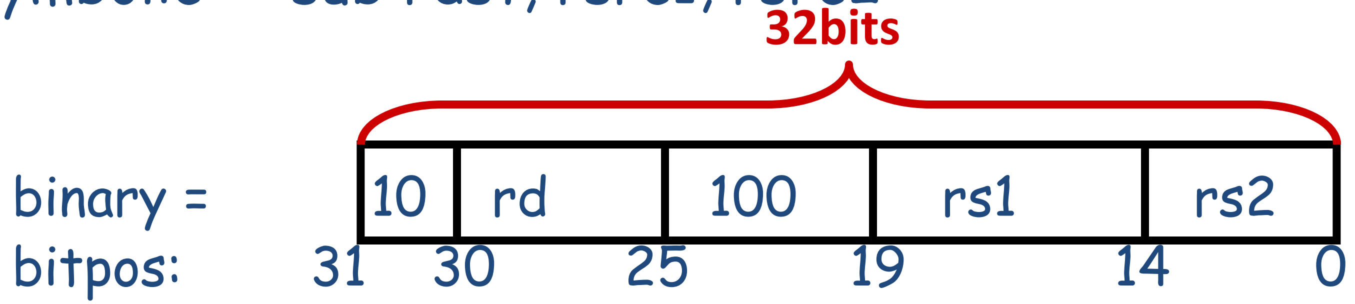
  Total runtime = code gen cost + cost of running code

How?

- Determine binary encoding of desired assembly instructions

  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)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)