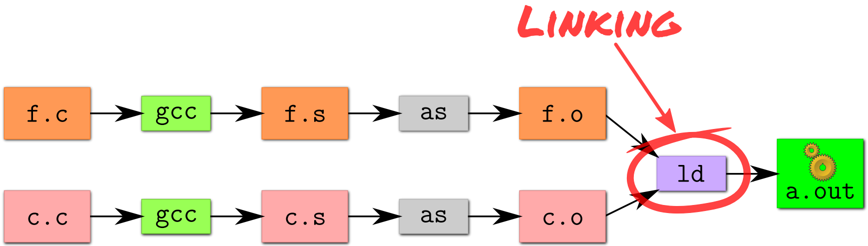
Today's Big Adventure



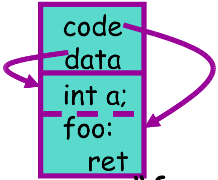
- **How to name and refer to things that don't exist yet**
- **How to merge separate name spaces into a cohesive whole**
- **More information**
  - The ELF standard
  - Run "`nm`," "`objdump`," and "`readelf`" on a few .o and a.out files.

Perspectives on memory contents

- **Programming language view:** `x += 1;  add $1, %eax`
  - Instructions: Specify operations to perform
  - Variables: Operands that can change over time
  - Constants: Operands that never change
- **Hardware view:**
  - executable: code, usually read-only
  - read only: constants (maybe one copy for all processes)
  - read/write: variables (each process needs own copy)
- **Need *addresses* to use data:**
  - Addresses locate things. Must update them when you move
  - Examples: linkers, garbage collectors, URL
- **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?



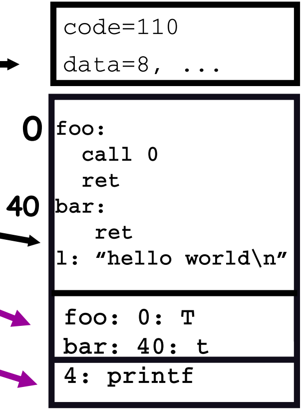
“foo.o”

- **Linker builds executables from object files:**

Header: code/data size, symtab offset

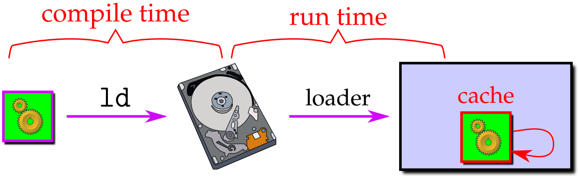
Object code: instructions and data gen'd by compiler

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



How is a program executed?

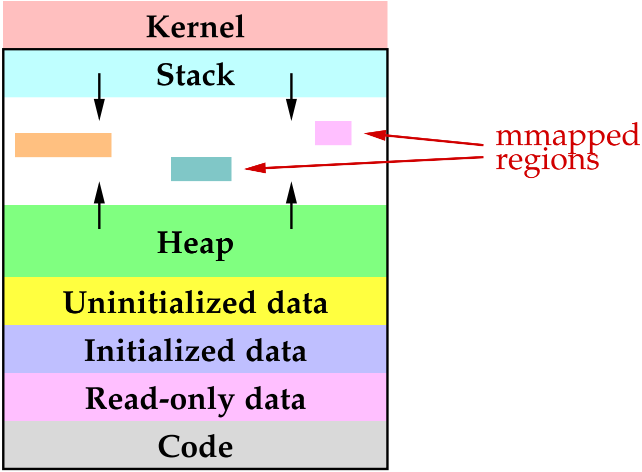
- **On Unix systems, read by "loader"**



  - Reads all code/data segments into buffer cache;
    Maps code (read only) and initialized data (r/w) into addr space
  - Or. . . fakes process state to look like paged out
- **Lots of optimizations happen in practice:**
  - 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

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: allocated and laid out at runtime by 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 at runtime (procedure calls), layout by compiler**
  - Names are relative off of stack (or frame) pointer
  - Managed by compiler (alloc on procedure entry, free on exit)
  - Linker not involved because name space entirely local: Compiler has enough information to build it.
- **Global data/code: alloc by compiler, layout by linker**
  - Compiler emits them and names with symbolic references
  - Linker lays them out and translates references

Example

- **Simple program has "`printf ("hello world\n");`"**
- **Compile with:** `cc -m32 -fno-builtin -S hello.c`
  - `-S` says don't run assembler (`-m32` is 32-bit x86 code)
- **Output in `hello.s` has symbolic reference to `printf`**

```
        .section        .rodata
.LC0:   .string "hello world\n"
        .text
.globl main
main:   ...
        subl    $4, %esp
        movl    $.LC0, (%esp)
        call    printf
```

- **Disassemble `.o` file with `objdump -d`:**

```
18: e8 fc ff ff ff      call 19 <main+0x19>
```

  - Jumps to PC - 4 = address of address within instruction

Linkers (Linkage editors)

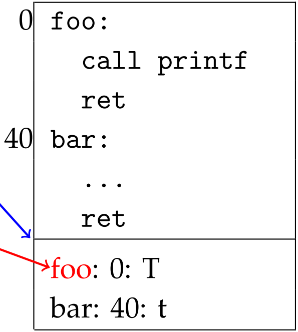
- **Unix: ld**
  - Usually hidden behind compiler
  - Run `gcc -v hello.c` to see ld or invoked (may see collect2)
- **Three functions:**
  - Collect together all pieces of a program
  - Coalesce like segments
  - Fix addresses of code and data so the program can run
- **Result: runnable program stored in new object file**
- **Why can't compiler do this?**
  - Limited world view: sees one file, rather than all files
- **Usually linkers don't rearrange segments, but can**
  - E.g., re-order instructions for fewer cache misses;
    remove routines that are never called from `a.out`

Simple linker: two passes needed

- **Pass 1:**
  - Coalesce like segments; arrange in non-overlapping memory
  - Read files' symbol tables, construct global symbol table with entry for every symbol used or defined
  - Compute virtual address of each segment (at start+offset)
- **Pass 2:**
  - Patch references 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

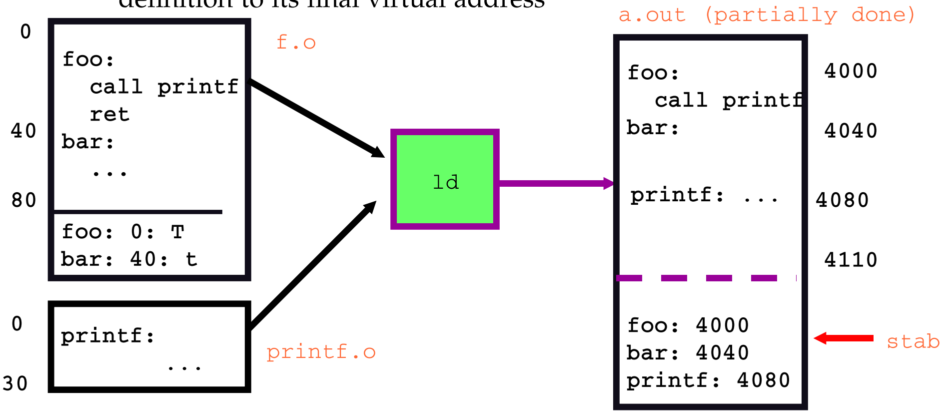
Where to put emitted objects?

- **Assember:**
  - 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
  - Routines/variables exported by file are recorded as global definitions
- **Simpler perspective:**
  - Code is in a big char array
  - Data is in another big char array
  - Assembler creates (object name, index) tuple for each interesting thing
  - Linker then merges all of these arrays



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



Where is everything?

- **How to call procedures or reference variables?**
  - E.g., call to `printf` needs a target addr
  - Assembler uses 0 or PC for address
  - Emits an external reference telling the linker the instruction's offset and the symbol it needs to be patched with

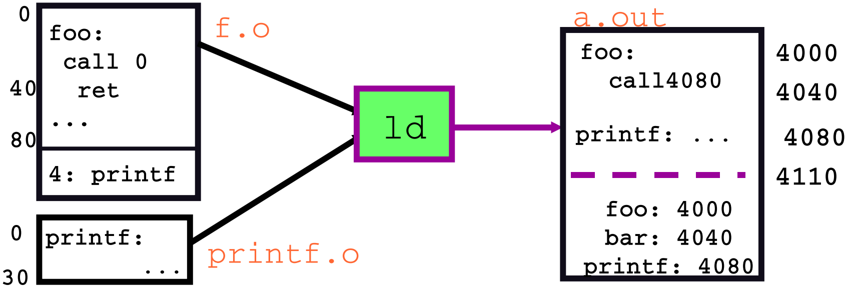
```
0   foo:
      pushl $.LC0
4     call -4
      ret
40  bar:
      ...
      ret
```

foo: 0: T	
bar: 40: t	
printf: 4	

- **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



Example: 2 modules and C lib

main.c

```
extern float sin();
extern int printf(), scanf();
float val = 0.0;
int main() {
  static float x = 0.0;
  printf("enter␣number:␣");
  scanf("%f", &x);
  printf("Sine␣is␣%f\n", val);
}
```

math.c

```
float sin(float x) {
  float tmp1, tmp2;
  static float res = 0.0;
  static float lastx = 0.0;
  if (x != lastx) {
    lastx = x;
    /* compute sin(x) */
  }
  return res;
}
```

libc

```
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)
8    val = call sin(x)      text
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)
4       lastx = x;   text
…       … compute sin(x)…
24   return res;
```

Pass 1: Linker reorganization

a.out:

```
     symbol table

0     val:
4     x:
8     res:
12    lastx:

16   main:
…      …
26     call  printf(val)
30   sin:
…      …
50     return res;       text
64   printf: …
80   scanf: …
```

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

"final" a.out:

Starting virtual addr: 4000

Offset	a.out	Virtual addr
	symbol table	
0	val:	4000
4	x:	4004
8	res:	4008
12	lastx: data	4012
16	main:	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	
	Text segment	
16	main:	4016
26	call 4064(4000)	4026
30	sin:	4030
50	return load 4008;	4050
64	printf:	4064
80	scanf:	4080
1000	Data segment	5000
	val: 0.0	
	x: 0.0	
	...	

virtual addr: 4016

Symbol table:

```
initialized data = 4000
uninitialized data = 4000
text = 4016
def: val @ 1000
def: x @ 1004
def: res @ 1008
def: main @ 14
def: sin @ 30
def: printf @ 64
def: scanf @ 80
```

Examining programs with nm

```
int uninitialized;
int initialized = 1;
const int constant = 2;
int main ()
{
  return 0;
}
```

```
$ nm a.out
...
0400400 T _start
04005bc R constant
0601008 W data_start
0601020 D initialized
04004b8 T main
0601028 B uninitialized
```

VA

symbol type

- `const` **variables of type** `R` **won't be written**
  - Note `constant` VA on same page as `main`
  - Share pages of read-only data just like text
- **Uninitialized data in "BSS" segment,** `B`
  - No actual contents in executable file
  - Goes in pages that the OS allocates zero-filled, on-demand

Examining programs with objdump

Note Load mem addr. and File off have same page alignment for easy mmapping

```
$ objdump -h a.out
a.out:     file format elf64-x86-64
Sections:
Idx Name     Size      VMA       LMA       File off  Algn
...
 12 .text    000001a8  00400400  00400400  00000400  2**4
             CONTENTS, ALLOC, LOAD, READONLY, CODE
...
 14 .rodata 00000008  004005b8  004005b8  000005b8  2**2
             CONTENTS, ALLOC, LOAD, READONLY, DATA
...
 17 .ctors   00000010  00600e18  00600e18  00000e18  2**3
             CONTENTS, ALLOC, LOAD, DATA
...
 23 .data    0000001c  00601008  00601008  00001008  2**3
             CONTENTS, ALLOC, LOAD, DATA
...
 24 .bss     0000000c  00601024  00601024  00001024  2**2
             ALLOC
...
```

No contents in file

Types of relocation

- **Place final address of symbol here**
  - Example: `int y, *x = &y;`
    `y` gets address in BSS, `x` in data segment, contains VA of `y`
  - Code example: `call printf` becomes
    `8048248:  e8 e3 09 00 00  call 8048c30 <printf>`
  - Binary encoding reflects computed VMA of `printf`
    (Note encoding of `call` argument is actually PC-relative)
- **Add address of symbol to contents of this location**
  - Used for record/struct offsets
  - Example: `struct queue { int type; void *head; } q;`
    `q.head = NULL` → `movl $0, q+4` → `movl $1, 0x804a01c`
- **Add diff between final and original seg to this location**
  - Segment was moved, “static” variables need to be relocated

Name mangling

Mangling not compatible across compiler versions

```
// C++
int foo (int a)
{
  return 0;
}

int foo (int a, int b)
{
  return 0;
}
```

```
% nm overload.o
0000000 T _Z3fooi
000000e T _Z3fooii
        U __gxx_personality_v0
```

Demangle names

```
% nm overload.o | c++filt
0000000 T foo(int)
000000e T foo(int, int)
        U __gxx_personality_v0
```

- **C++ can have many functions with the same name**
- **Compiler therefore *mangles* symbols**
  - Makes a unique name for each function
  - Also used for methods/namespaces (`obj::fn`), template instantiations, & special functions such as `operator new`

Initialization and destruction

```
// C++
int a_foo_exists;
struct foo_t {
  foo_t () {
    a_foo_exists = 1;
  }
};
foo_t foo;
```

- **Initializers run before main**
  - Mechanism is platform-specific
- **Example implementation:**
  - Compiler emits static function in each file running initializers
  - Wrap linker with `collect2` program that generates `__main` function calling all such functions
  - Compiler inserts call to `__main` when compiling real `main`

```
% cc -S -o- ctor.C | c++filt
...
        .text
        .align 2
__static_initialization_and_destruction_0(int, int):
...
        call    foo_t::foo_t()
```

Other information in executables

```
// C++
struct foo_t {
  ~foo_t() {/*...*/}
  except() { throw 0; }
};
void fn ()
{
  foo_t foo;
  foo.except();
  /* ...  */
}
```

- **Throwing exceptions destroys automatic variables**
- **Must find all such variables**
  - In all procedures' call frames until exception caught
  - All variables of types with non-trivial destructors
- **Record info in special sections**

- **Executables can include debug info (compile w. -g)**
  - What source line does each binary instruction correspond to?

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

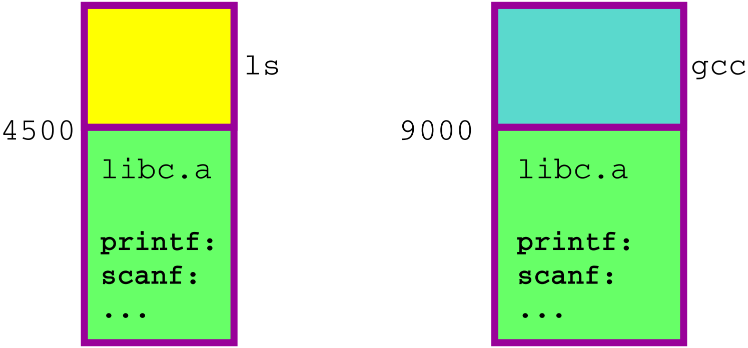
```
foo:
  call puts
```

```
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?

Variation 1: Static shared libraries

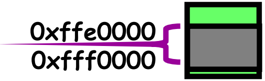
- **Observation: everyone links in standard libraries (libc.a.), these libs consume space in every executable.**

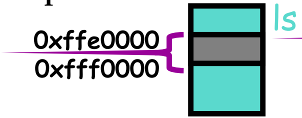


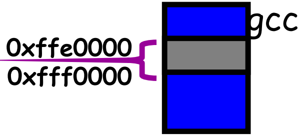
- **Insight: we can have a single copy on disk if we don't actually include libc code in executable**

Static shared libraries

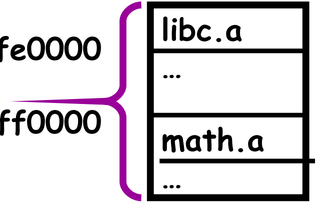
- Define a "shared library segment" at same address in every program's address space







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



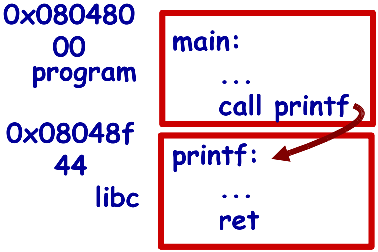
- Linker links program against lib (why?) but does not bring in actual code
- Loader marks shared lib region as unreadable
- When process calls lib code, seg faults: embedded linker brings in lib code from known place & maps it in.
- Now different running programs can share code!

Variation 2: Dynamic shared libs

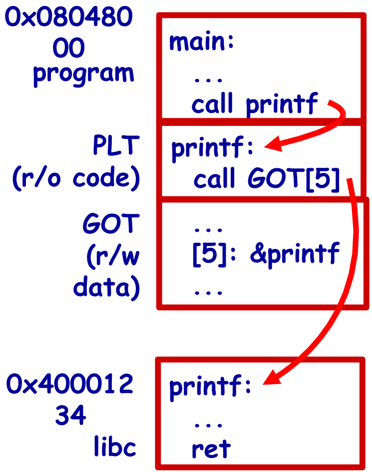
- **Static shared libraries require system-wide pre-allocation of address space**
  - Clumsy, inconvenient
  - What if a library gets too big for its space?
  - Can space ever be reused?
- **Solution: Dynamic shared libraries**
  - Let any library be loaded at any VA
  - New problem: Linker won't know what names are valid
  - Solution: stub library
  - New problem: How to call functions if their position may vary?
  - Solution: next page…

Position-independent code

- Code must be able to run anywhere in virtual mem
- Runtime linking would prevent code sharing, so…
- Add a level of indirection!



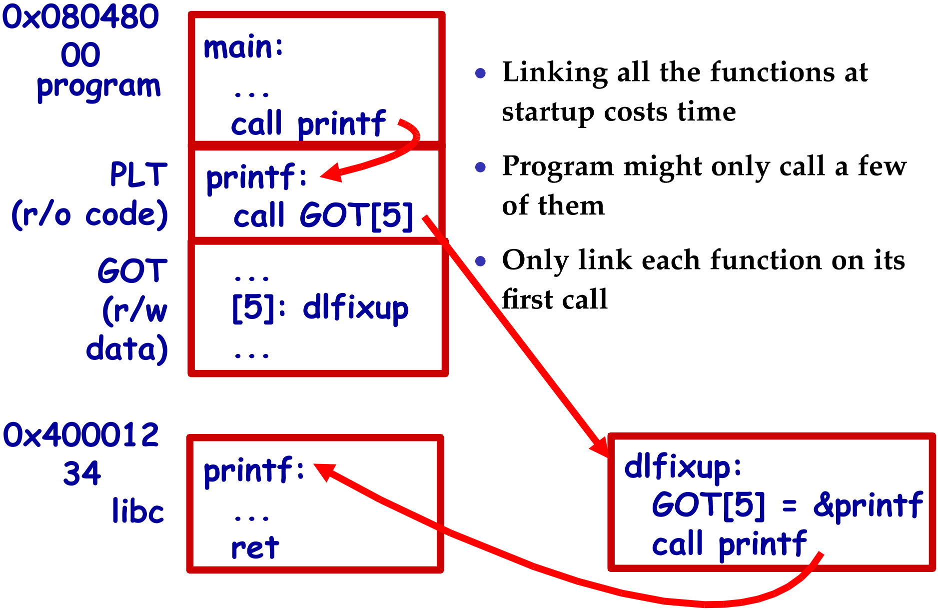
Static Libraries



Dynamic Shared Libraries

Lazy dynamic linking

- Linking all the functions at startup costs time
- Program might only call a few of them
- Only link each function on its first call



ELF

- **Today many systems use ELF as a binary format**
- **Every ELF file has an *ELF header* (`readelf -h` *file*)**
- **Files ready to be run by OS have *program headers***
  - Examine with `readelf -l` *file*
  - Goes near beginning of file; says where to load what and how
- **Files that need to be linked have *section headers***
  - Examine with `readelf -S` *file*
  - Goes at end of file (may not need to be mapped in)

Dynamic linking with ELF

- **Every dynamically linked executable needs an *interpreter***
  - Embedded as string in special `.interp` section
  - `readelf -p .interp /bin/ls` → `/lib64/ld-linux-x86-64.so.2`
  - So all the kernel has to do is run `ld-linux`
- `dlfixup` **uses hash table to find symbols when needed**
- **Hash table lookups can be quite expensive [Drepper]**
  - E.g., big programs like OpenOffice very slow to start
  - Solution 1: Use a better hash function
    - ▷ linux added `.gnu.hash` section, later removed `.hash` sections
  - Solution 2: Export fewer symbols (it is now fashionable to use:
    - ▷ `gcc -fvisibility=hidden` (keep symbols local to DSO)
    - ▷ `#pragma GCC visibility push(hidden)/visibility pop`
    - ▷ `__attribute__(visibility("default"))`, (override for a symbol)

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, or qemu vs. bochs.
  - 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 instructions**

```
SPARC: sub instruction
       symbolic = "sub rdst, rsrc1, rsrc2"
                                    32bits
       binary =   10   rd     100     rs1      rs2
       bit pos:  31 30     25      19       14     0
```

- **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)();
```

Linking and security

```
void fn ()
{
  char buf[80];
  gets (buf);
  /* ...  */
}
```

1. **Attacker puts code in buf**
   - Overwrites return address to jump to code
2. **Attacker puts shell command above buf**
   - Overwrites return address so function "returns" to `system` function in libc

- **People try to address problem with linker**
- **W^X: No memory both writable and executable**
  - Prevents 1 but not 2, must be disabled for jits
- **Address space randomization**
  - Makes attack #2 a little harder, not impossible
- **Also address with compiler (stack protector)**

Linking Summary

- **Compiler/Assembler: 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`)