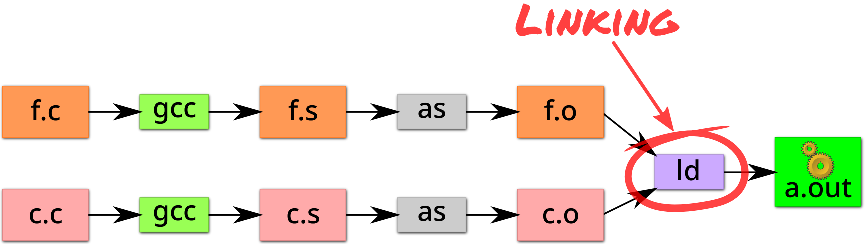
Administrivia

- **Lab 2 due Friday**
- **Midterm review section Friday**
- **Midterm exam in class next Monday Feb. 12**
  - Open note, but no textbook or electronic devices
  - Bring lecture note printouts
  - SCPD must register exam monitor or show up in person
    (no need to request permission to show up in person)

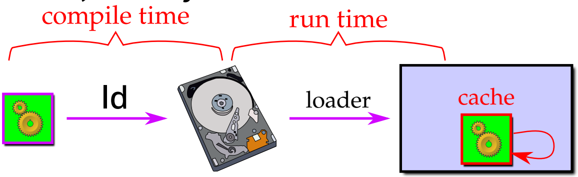
Today's Big Adventure



- **How to name and refer to things that don't exist yet**
- **How to merge separate name spaces into a cohesive whole**
- **More information:**
  - How to write shared libraries
  - Run "`nm`," "`objdump`," and "`readelf`" on a few .o and a.out files.
  - The ELF standard
  - Examine `/usr/include/elf.h`

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

x86 Assembly syntax

- **Linux uses AT&T assembler syntax – places destination last**
  - Be aware that *intel syntax* (used in manual) places destination first
- **Types of operand available:**
  - Registers start with "%" – `movl %edx,%eax`
  - Immediate values (constants) prefixed by "$" – `movl $0xff,%edx`
  - (%*reg*) is value at address in register *reg* – `movl (%edi),%eax`
  - *n*(%*reg*) is value at address in (register *reg*)+*n* – `movl 8(%ebp),%eax`
  - *%*reg* in an indirection through *reg* – `call *%eax`
  - Everything else is an address – `movl var,%eax; call printf`
- **Some heavily used instructions**
  - `movl` – moves (copies) value from source to destination
  - `pushl`/`popl` – pushes/pops value on stack
  - `call` – pushes next instruction address to stack and jumps to target
  - `ret` – pops address of stack and jumps to it
  - `leave` – equivalent to `movl %ebp,%esp; popl %ebp`

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

Running example: hello program

- **Hello program**
  - Write friendly greeting to terminal
  - Exit cleanly
- **Every programming language addresses this problem**

[demo]

Running example: hello program

- **Hello program**
  - Write friendly greeting to terminal
  - Exit cleanly
- **Every programming language addresses this problem**
- **Concept should be familiar if you took 106B:**

```
int
main()
{
    cout << "Hello, world!" << endl;
}
```

- **Today's lecture: 80 minutes on hello world**

Hello world – CS212-style

```
#include <sys/syscall.h>
int my_errno;
const char greeting[] = "hello world\n";

int my_write(int fd, const void *buf, size_t len)
{
  int ret;
  asm volatile ("int $0x80" : "=a" (ret)
                : "0" (SYS_write),
                  "b" (fd), "c" (buf), "d" (len)
                : "memory");
  if (ret < 0) {
    my_errno = -ret;
    return -1;
  }
  return ret;
}

int main() { my_write (1, greeting, my_strlen(greeting)); }
```

Examining `hello1.s`

- **Grab the source and try it yourself**
  - `tar xzf /afs/ir.stanford.edu/class/cs212/hello.tar.gz`
- `gcc -S hello1.c` **produces assembly output in** `hello1.s`
- **Check the definitions of** `my_errno`, `greeting`, `main`, `my_write`
- `.globl` **symbol makes *symbol* global**
- **Sections of** `hello1.s` **are directed to various segments**
  - `.text` says put following contents into text segment
  - `.data`, `.rodata` says to put into data or read-only data
  - `.comm` *symbol,size,align* declares *symbol* and allows multiple definitions (like C but not C++, now requires `-fcommon` flag)
- **See how function calls push arguments to stack, then pop**

```
pushl   $greeting  # Argument to my_strlen is greeting
call    my_strlen  # Make the call (length now in %eax)
addl    $4, %esp   # Must pop greeting back off stack
```

Disassembling `hello1`

```
 my_write (1, greeting, my_strlen(greeting));
8049208:   68 08 a0 04 08       push   $0x804a008
804920d:   e8 93 ff ff ff       call   80491a5 <my_strlen>
8049212:   83 c4 04             add    $0x4,%esp
8049215:   50                   push   %eax
8049216:   68 08 a0 04 08       push   $0x804a008
804921b:   6a 01                push   $0x1
804921d:   e8 aa ff ff ff       call   80491cc <my_write>
8049222:   83 c4 0c             add    $0xc,%esp
```

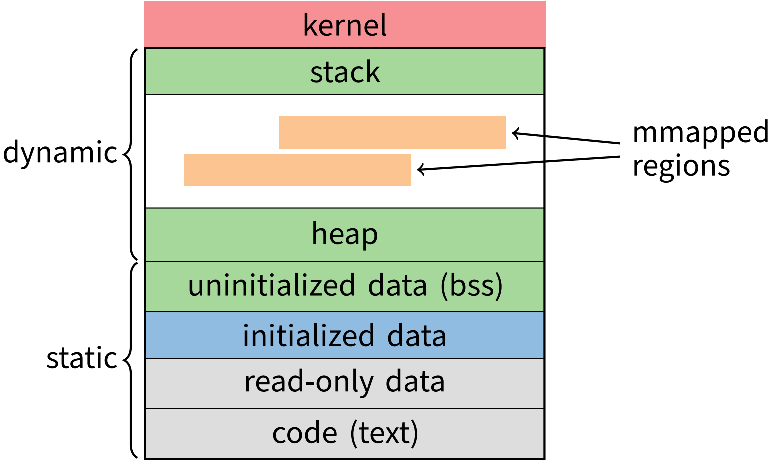
- **Disassemble from shell with** `objdump -Sr hello1`
- **Note** `push` **encodes address of greeting (0x804a008)**
- **Offsets in** `call` **instructions: 0xffffff93 = -109, 0xffffffaa = -86**
  - Binary encoding takes offset relative to next instruction

How is a process specified?

```
$ readelf -h hello1
ELF Header:
  ...
  Entry point address:               0x8049030
  Start of program headers:          52 (bytes into file)
  Start of section headers:          14968 (bytes into file)
  Number of program headers:         8
  Number of section headers:         23
  Section header string table index: 22
```

- **Executable files are the linker/loader interface. Must tell OS:**
  - What is code? What is data? Where should they live?
  - This is part of the purpose of the ELF standard
- **Every ELF file starts with ELF an *header***
  - Specifies *entry point* virtual address at which to start executing
  - But how should the loader set up memory?

Recall what process memory looks like



- **Address space divided into “segments”**
  - Text, read-only data, data, bss, heap (dynamic data), and stack
  - Recall gcc told assembler in which segments to put what contents

Who builds what?

- **Heap: allocated and laid out at runtime by malloc**
  - Namespace constructed dynamically, managed by *programmer* (names stored in pointers, and organized using data structures)
  - Compiler, linker not involved other than saying where it can start
- **Stack: allocated at runtime (func. 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 namespace entirely local: Compiler has enough information to build it.
- **Global data/code: allocated by compiler, layout by *linker***
  - Compiler emits them and names with symbolic references
  - Linker lays them out and translates references
- **Mmapped regions: Managed by programmer or linker**
  - Some programs directly call `mmap`; dynamic linker uses it, too

ELF program header

```
$ readelf -l hello1
Program Headers:
  Type      Offset    VirtAddr    PhysAddr    FileSiz MemSiz  Flg Align
  LOAD      0x001000 0x08049000 0x08049000 0x00304 0x00304 R E 0x1000
  LOAD      0x002000 0x0804a000 0x0804a000 0x00158 0x00158 R   0x1000
  LOAD      0x002ff8 0x0804bff8 0x0804bff8 0x0001c 0x0003c RW  0x1000
  ...
 Section to Segment mapping:
  Segment Sections...
   01      ... .text ...
   02      .rodata ...
   03      ... .data .bss
```

- **For executables, the ELF header points to a *program header***
  - Says what segments of file to map where, with what permissions
- **Segment 03 has shorter file size then memory size**
  - Only 0x1c bytes must be read into memory from file
  - Remaining 0x20 bytes constitute the .bss
- **Who creates the program header? The linker**

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?**
- **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`

Linkers (Linkage editors)

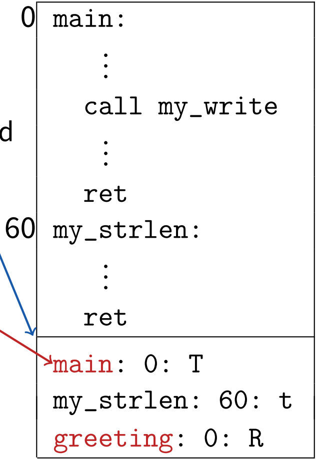
- **Unix: ld**
  - Usually hidden behind compiler
  - Run `gcc -v hello.c` to see ld or invoked (may see collect2)
- **Three functions:**
  - Collect together all pieces of a program
  - Coalesce like segments
  - Fix addresses of code and data so the program can run
- **Result: runnable program stored 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 each segment 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 byte array
  - Data is in another big byte array
  - Assembler creates (object name, index) tuple for each interesting thing
  - Linker then merges all of these arrays



Object files

```
$ objdump -Sr hello2.o
  ...
  48:   50                          push   %eax
  49:   68 00 00 00 00              push   $0x0
                            4a: R_386_32     greeting
  4e:   6a 01                       push   $0x1
  50:   e8 fc ff ff ff              call   51 <main+0x2a>
                            51: R_386_PC32  my_write
  55:   83 c4 10                    add    $0x10,%esp
```

- **Let's create two-file program `hello2` with `my_write` in separate file**
  - Compiler and assembler can't possibly know final addresses
- **Notice `push` uses 0 as address of `greeting`**
- **And `call` uses -4 as address of `my_write`—why?**

Object files

```
$ objdump -Sr hello2.o
  ...
  48:   50                              push   %eax
  49:   68 00 00 00 00                  push   $0x0
                            4a: R_386_32      greeting
  4e:   6a 01                           push   $0x1
  50:   e8 fc ff ff ff                  call   51 <main+0x2a>
                            51: R_386_PC32   my_write
  55:   83 c4 10                        add    $0x10,%esp
```

- **Let's create two-file program `hello2` with `my_write` in separate file**
  - Compiler and assembler can't possibly know final addresses
- **Notice `push` uses 0 as address of `greeting`**
- **And `call` uses -4 as address of `my_write`—why?**
  - Target (sitting at offset 51 in text) encoded relative to next instruction (`add` at offset 55)

Where is everything?

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

```
0   main:
       ⋮
49     pushl $0x0
4e     pushl $0x1
50     call -4
       ⋮
    ---------------------
    main: 0: T
    my_strlen: 40: t
    ---------------------
    greeting:  4a
    my_write:  51
```

- **At link time the linker patches every reference**

Relocations

```
$ readelf -r hello2.o
    ⋮
 Offset     Info    Type            Sym.Value  Sym. Name
00000039  00000801 R_386_32          00000000   greeting
0000004a  00000801 R_386_32          00000000   greeting
00000051  00000a02 R_386_PC32        00000000   my_write
    ⋮
```

- **Object file stores list of required relocations**
  - `R_386_32` says add symbol value to value already in file (often 0)
  - `R_386_PC32` says add difference between symbol value and patch location to value already in file (often -4 for `call`)
  - Info encodes type and index of symbol value to use for patch

ELF sections

```
$ readelf -S hello2.o
  [Nr] Name         Type         Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]              NULL         00000000 000000 000000 00      0   0  0
  [ 1] .text        PROGBITS     00000000 000034 0000a4 00  AX  0   0  1
  [ 2] .rel.text    REL          00000000 0005f8 000018 08   I 20   1  4
  [ 3] .data        PROGBITS     00000000 0000d8 000000 00  WA  0   0  1
  [ 4] .bss         NOBITS       00000000 0000d8 000000 00  WA  0   0  1
  [ 5] .rodata      PROGBITS     00000000 0000d8 00000d 00   A  0   0  4
   ⋮
  [20] .symtab      SYMTAB       00000000 0004f0 0000d0 10     21   9  4
  [21] .strtab      STRTAB       00000000 0005c0 000038 00      0   0  1
```

- **Memory segments have corresponding PROGBITS file segments**
- **But relocations and symbol tables reside in segments, too**
- **Segments can be arrays of fixed-size data structures**
  - So strings referenced as offsets into special string segments
- **Remember ELF header had section header string table index**
  - That's so you can interpret names in section header

Symbol table

```
$ readelf -s hello2.o
   Num:    Value  Size Type    Bind   Vis      Ndx Name
              ⋮
     3: 00000000    39 FUNC    LOCAL  DEFAULT    1 my_strlen
              ⋮
     9: 00000000    13 OBJECT  GLOBAL DEFAULT    5 greeting
    10: 00000027    62 FUNC    GLOBAL DEFAULT    1 main
    11: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND my_write
              ⋮
```

- **Lists all global, exported symbols**
  - Sometimes local ones, too, for debugging (e.g., `my_strlen`)
- **Each symbol has an offset in a particular section number**
  - On previous slide, 1 = `.text`, 5 = `.rodata`
  - Special undefined section 0 means need symbol from other file

How to lay out emitted objects?

- **At link time, linker first:**
  - Coalesces all like segments (e.g., all `.text`, `.rodata`) from all files
  - 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
- **Then in a second phase:**
  - Ensure each symbol has exactly 1 definition (except weak symbols, when compiling with `-fcommon`)
  - For each relocation:
    - Look up referenced symbol's virtual address in symbol table
    - Fix reference to reflect address of referenced symbol

What is a library?

- **A static library is just a collection of `.o` files**
- **Bind them together with `ar` program, much like `tar`**
  - E.g., `ar cr libmylib.a obj1.o obj2.o obj3.o`
  - On many OSes, run `ranlib libmylib.a` (to build index)
- **You can also list (`t`) and extract (`x`) files**
  - E.g., try: `ar tv /usr/lib/libc.a`
- **When linking a `.a` (archive) file, linker only pulls in needed files**
  - Ensures resulting executable can be smaller than big library
- `readelf` **will operate on every archive member (unweildy)**
  - But often convenient to disassemble with
    `objdump -d /usr/lib/libc.a`

Examining programs with nm

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

VA

symbol type

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

- **If don't need full `readelf`, can use `nm` (`nm -D` on shared objects)**
  - Handy `-o` flag prints file, useful with `grep`
- **`R` means read-only data (`.rodata` in elf)**
  - Note `constant` VA on same page as `main`
  - Share pages of read-only data just like text
- **`B` means uninitialized data in "BSS"**
- **Lower-case letters correspond to local symbols (static in C)**

Examining sections 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

- **Another portable alternative to** `readelf`

Name mangling

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

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

Mangling not compatible across compiler versions

```
% 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**
- **During exception, must find**
  - All such variables with non-trivial destructors
  - In all procedures' call frames until exception caught
- **Record info in special sections**

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

Dynamic (runtime) linking (`hello3.c`)

```
#include <dlfcn.h>
int main(int argc, char **argv, char **envp)
{
  size_t (*my_strlen)(const char *p);
  int (*my_write)(int, const void *, size_t);
  void *handle = dlopen("dest/libmy.so", RTLD_LAZY);
  if (!handle
      || !(my_strlen = dlsym(handle, "my_strlen"))
      || !(my_write = dlsym(handle, "my_write")))
    return 1;
  return my_write (1, greeting, my_strlen(greeting)) < 0;
}
```

- **Link time isn't special, can link at runtime too**
  - Get code (e.g., plugins) not available when program compiled
- **Issues:**
  - How can behavior differ compared to static linking?
  - Where to get unresolved symbols (e.g., `my_write`) from?
  - How does `my_write` know its own addresses (e.g., for `my_errno`)?

Dynamic linking (continued)

- **How can behavior differ compared to static linking?**
  - Runtime failure (can't find file, doesn't contain symbols)
  - No type checking of functions, variables
- **Where to get unresolved symbols (e.g., `my_write`) from?**
  - `dlsym` must parse ELF file to find symbols
- **How does `my_write` know its own addresses?**

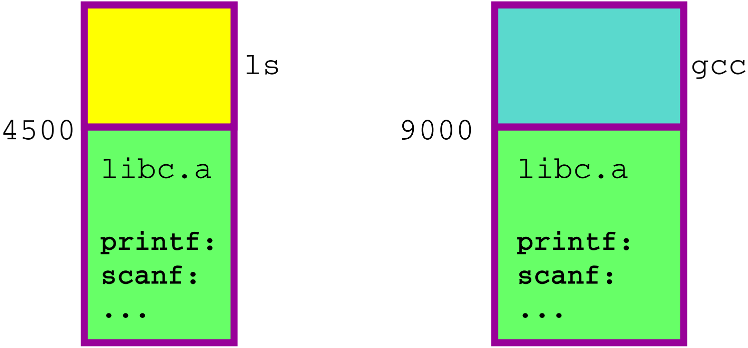
```
$ readelf -r dest/libmy.so

Relocation section '.rel.dyn' at offset 0x20c contains 1 entry:
 Offset     Info    Type            Sym.Value  Sym. Name
00003ffc  00000106 R_386_GLOB_DAT    0000400c   my_errno
```

  - `dlopen`, too, must parse ELF to patch relocations

Static shared libraries

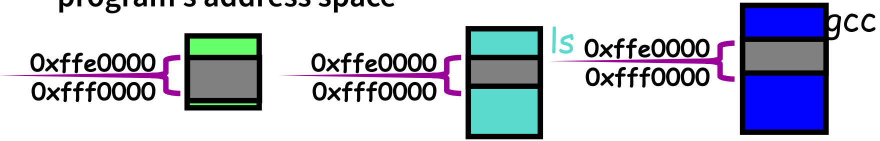
- **Observation: everyone links in standard libraries (libc.a.), these libs consume space in every executable.**



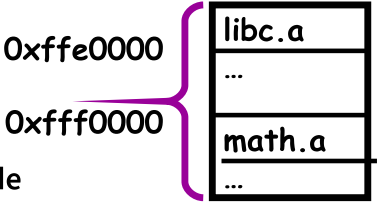
- **Insight: we can have a single copy on disk if we don't actually include libc code in executable**

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



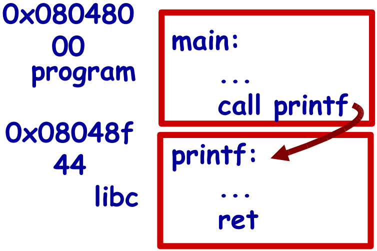
- Linker links program against lib (why?) but does not bring in actual code
- Loader marks shared lib region as unreadable
- When process calls lib code, seg faults: embedded linker brings in lib code from known place & maps it in.
- Now different running programs can share code!

Dynamic shared libraries

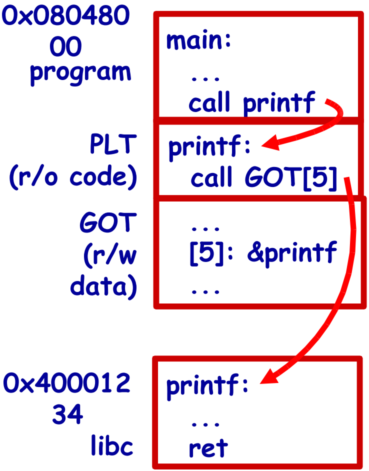
- **Static shared libraries require system-wide pre-allocation of address space**
  - Clumsy, inconvenient
  - What if a library gets too big for its space? (fragmentation)
  - Can't upgrade libraries w/o relinking applications
  - Can space ever be reused?
- **Solution: Dynamic shared libraries**
  - Combine shared library and dynamic linking ideas
  - Any library can be loaded at any VA, chosen at runtime
- **New problem: Linker won't know what names are valid**
  - Solution: stub library
- **New problem: How to call functions whose position varies?**
  - 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!

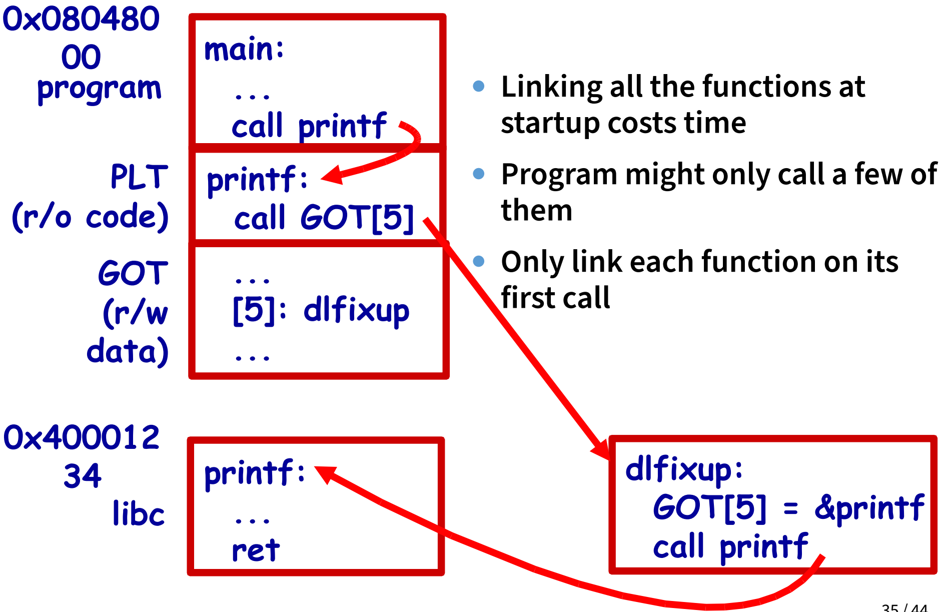


Static Libraries



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

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. 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)

Dynamic shared library example: `hello4`

```
$ objdump -Sr hello4
     ⋮
08049030 <my_write@plt>:
 8049030:       ff 25 0c c0 04 08       jmp    *0x804c00c
 8049036:       68 00 00 00 00          push   $0x0
 804903b:       e9 e0 ff ff ff          jmp    8049020 <.plt>

08049040 <my_strlen@plt>:
 8049040:       ff 25 10 c0 04 08       jmp    *0x804c010
 8049046:       68 08 00 00 00          push   $0x8
 804904b:       e9 d0 ff ff ff          jmp    8049020 <.plt>
     ⋮
 804917a:       68 08 a0 04 08          push   $0x804a008
 804917f:       e8 bc fe ff ff          call   8049040 <my_strlen@plt>
```

- **`0x804c00c` and `0x804c010` initially point to next instruction**
  - Calls `dlfixup` with relocation index
  - Note second `jmp` of each entry goes to 0th PLT entry, which jumps to `dlfixup`

hello4 relocations

```
$ readelf -r hello4
Relocation section '.rel.plt' at offset 0x314 contains 2 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
0804c00c  00000107 R_386_JUMP_SLOT   00000000   my_write
0804c010  00000507 R_386_JUMP_SLOT   00000000   my_strlen
```

- **PLT = *procedure linkage table* on last slide**
  - Small 16 byte snippets, read-only executable code
- `dlfixup` **Knows how to parse relocations, symbol table**
  - Looks for symbols by name in hash tables of shared libraries
- `my_write` **&** `my_strlen` **are pointers in *global offset table* (GOT)**
  - GOT non-executable, read-write (so `dlfixup` can fix up)
- **Note** `hello4` **knows address of** `greeting`**, PLT, and GOT**
  - How does a shared object (`libmy.so`) find these?
  - PLT is okay because calls are relative
  - In PIC, compiler reserves one register `%ebx` for GOT address

hello4 shared object contents

mywrite.c

```c
int my_errno;
int my_write(int fd, const void *buf, size_t len) {
  int ret;
  asm volatile (/* ... */);
  if (ret < 0) {
    my_errno = -ret;
    return -1;
  }
  return ret;
}
```

mywrite.s

```
negl  %eax
movl  %eax, my_errno
```

mywrite-pic.s

```
negl  %eax
movl  %eax, %edx
movl  my_errno@GOT(%ebx), %eax
movl  %edx, (%eax)
```

How does %ebx get set?

mywrite-pic.s

```
my_write:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ebx
        subl    $16, %esp
        call    __x86.get_pc_thunk.bx
        addl    $_GLOBAL_OFFSET_TABLE_, %ebx
          ⋮
__x86.get_pc_thunk.bx:
        movl    (%esp), %ebx
        ret
```

```
$ readelf -r .libs/mywrite.o
 Offset     Info    Type            Sym.Value  Sym. Name
00000008  00000a02 R_386_PC32        00000000   __x86.get_pc_thunk.bx
0000000e  00000b0a R_386_GOTPC       00000000   _GLOBAL_OFFSET_TABLE_
00000036  0000082b R_386_GOT32X      00000000   my_errno
```

Linking and security

```c
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
  - Leads to position-independent executable, compiled `-fpie` and linked `-pie`—like PIC for executables
- **Also address with compiler (stack protector, CFI)**

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`)

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 Javascript compiler, 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;
...
```

- Use `mprotect` to disable W^X
- Jump to the address of the buffer: `((int (*)())code)();`