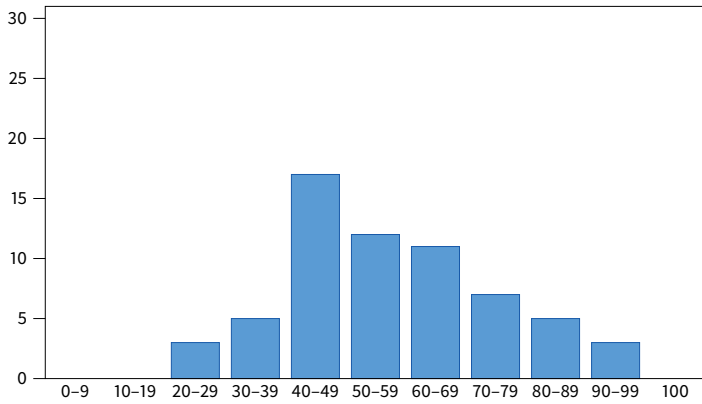


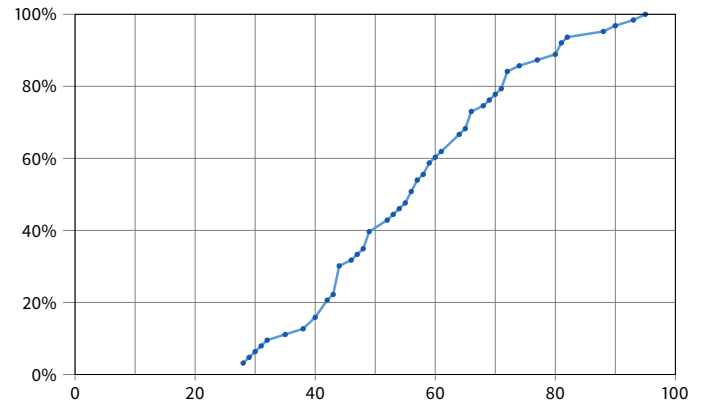
## Midterm results



- Mean: 57, median: 56

1 / 45

## Midterm results



- Systems students should insist on a CDF!

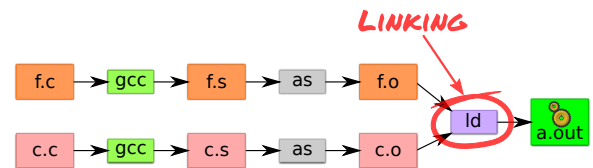
1 / 45

## Reminder

- Lab 3 section this Friday, 10:30pm Gates B-01

2 / 45

## Today's Big Adventure

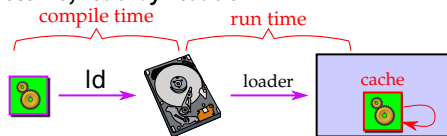


- How to name and refer to things that don't exist yet
- How to merge separate name spaces into a cohesive whole
- More information:
  - How to write shared libraries
  - Run "nm," "objdump," and "readelf" on a few .o and a.out files.
  - The ELF standard
  - Examine `/usr/include/elf.h`

3 / 45

## 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

4 / 45

## 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`

5 / 45

## 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

6 / 45

## Running example: hello program

- **Hello program**
  - Write friendly greeting to terminal
  - Exit cleanly
- **Microtechnology and programming language in today's computers ideally suited to solve this problem**

[demo]

7 / 45

## Running example: hello program

- **Hello program**
  - Write friendly greeting to terminal
  - Exit cleanly
- **Microtechnology and programming language in today's computers ideally suited to solve this problem**
- **Concept should be familiar if you took 106B:**

```
int main() {
    cout << "Hello, world!" << endl;
    return 0;
}
```
- **Today's lecture: 80 minutes on hello world**

7 / 45

## Hello world – CS140-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)); }
```

8 / 45

## Examining `hello1.s`

- **Watching video? Grab the source and try it yourself**
- `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
  - `.bss` is zero-initialized data (specify size, not value)
  - `.comm` *symbol, size, align* declares *symbol* and allows multiple definitions (like C but not C++)
- **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
```

9 / 45

## Disassembling `hello1`

```
my_write (1, greeting, my_strlen(greeting));
80482d0: 68 c0 83 04 08    push  $0x80483c0
80482d5: e8 92 ff ff ff   call  804826c <my_strlen>
80482da: 83 c4 04         add   $0x4,%esp
80482dd: 50              push  %eax
80482de: 68 c0 83 04 08    push  $0x80483c0
80482e3: 6a 01          push  $0x1
80482e5: e8 a9 ff ff ff   call  8048293 <my_write>
80482ea: 83 c4 0c         add   $0xc,%esp
```

- **Disassemble from shell with `objdump -Sr hello1`**
- **Offsets in `call` instructions: `0xffffffff92 = -110`, `0xffffffffa9 = -87`**
  - Binary encoding takes offset relative to next instruction
- **Note push encodes address of greeting (`0x80483c0`)**

10 / 45

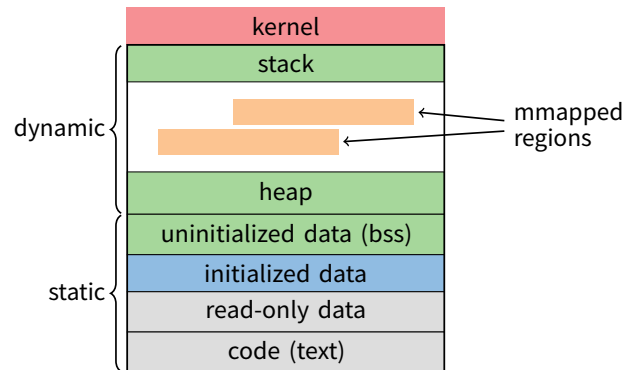
## How is a process specified?

```
$ readelf -h hello1
ELF Header:
...
Entry point address:      0x8048120
Start of program headers:  52 (bytes into file)
Number of program headers: 4
Start of section headers: 6852 (bytes into file)
Number of section headers: 22
Section header string table index: 21
```

- **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?

11 / 45

## 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

12 / 45

## 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

13 / 45

## ELF program header

```
$ readelf -l hello1
Program Headers:
Type   Offset  VirtAddr  PhysAddr  FileSiz MemSiz  Flg Align
LOAD   0x000000 0x08048000 0x08048000 0x004f4 0x004f4 R E 0x1000
LOAD   0x000ff8 0x08049ff8 0x08049ff8 0x0001c 0x0003c RW 0x1000
...
Section to Segment mapping:
Segment Sections...
00    ... .text .rodata ...
01    ... .data .bss
...
```

- **For executables, the ELF header points to a *program header***
  - Says what segments of file to map where, with what permissions
- **Segment 01 has shorter file size than 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**

14 / 45

## 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

15 / 45

## 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

15 / 45

## 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

16 / 45

## Where to put emitted objects?

- **Assembler:**
  - 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 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

0	main:
:	:
:	call my_write
:	:
:	ret
60	my_strlen:
:	:
:	ret
main: 0: T	
my_strlen: 60: t	
greeting: 0: R	

17 / 45

## 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?**

18 / 45

## 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)

18 / 45

## 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**

19 / 45

## Relocations

```
$ readelf -r hello2.o
:
Offset      Info      Type          Sym. Value  Sym. Name
00000039    00000f01  R_386_32      00000000    greeting
0000004a    00000f01  R_386_32      00000000    greeting
00000051    00001102  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

20 / 45

## 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 0005b8 000018 08  I 18  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
...
[18] .symtab     SYMTAB       00000000 000450 000130 10   19 15  4
```

- 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

21 / 45

## Symbol table

```
$ readelf -s hello2.o
Num:   Value Size Type   Bind  Vis      Ndx Name
      :
      :
      5: 00000000 39 FUNC   LOCAL  DEFAULT  1 my_strlen
      :
      :
     15: 00000000 13 OBJECT GLOBAL DEFAULT  5 greeting
     16: 00000027 62 FUNC   GLOBAL DEFAULT  1 main
     17: 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

22 / 45

## 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 syms.)
  - For each relocation:
    - ▷ Look up referenced symbol's virtual address in symbol table
    - ▷ Fix reference to reflect address of referenced symbol

23 / 45

## 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

24 / 45

## 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

- If don't need full readelf, can use nm (nm -D on shared objects)
  - Handy -o flag prints file, handy 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

25 / 45

## Examining sections with objdump

```
$ 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
```

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

No contents in file

- Another portable alternative to readelf

26 / 45

## Name mangling

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

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

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

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

Mangling not compatible across compiler versions

- 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

27 / 45

## 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()
```

28 / 45

## 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?

29 / 45

## 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)?

30 / 45

## 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

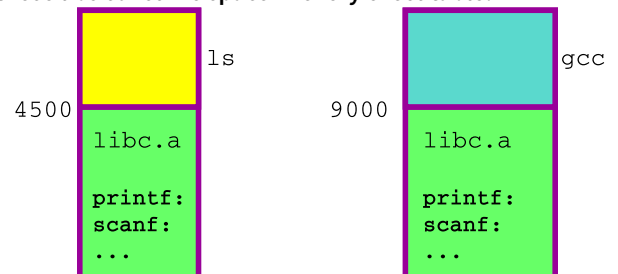
Relocation section '.rel.dyn' at offset 0x204 contains 1 entries:
Offset      Info      Type           Sym.Value  Sym. Name
000013bc    00000306  R_386_GLOB_DAT 000013cc   my_errno
```

- dlopen, too, must parse ELF to patch relocations

31 / 45

## Static shared libraries

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

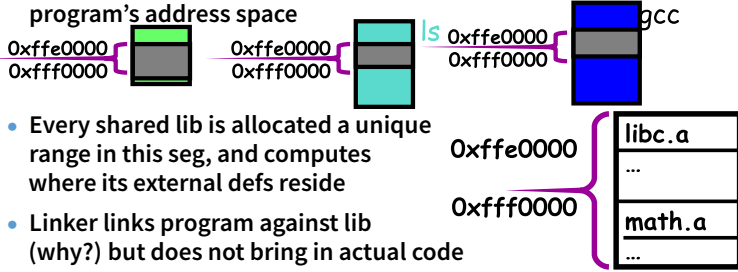


- Insight: we can have a single copy on disk if we don't actually include libc code in executable

32 / 45

## 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!



33 / 45

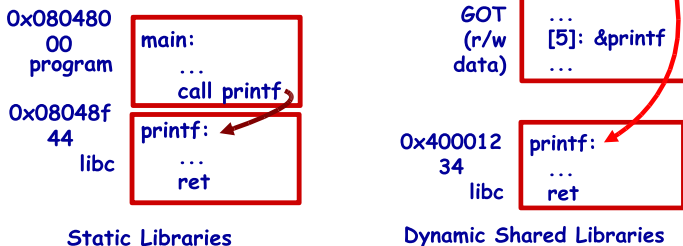
## Dynamic shared libraries

- 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 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...

34 / 45

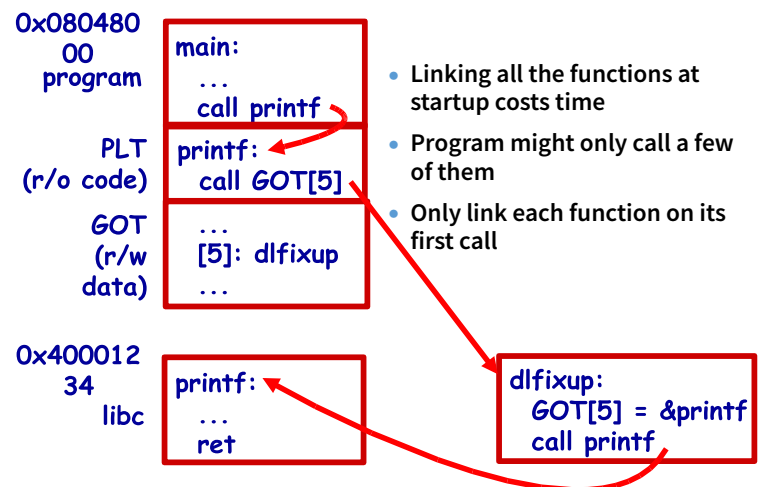
## Position-independent code

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



35 / 45

## Lazy dynamic linking



36 / 45

## 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)

37 / 45

## Dynamic shared library example: hello4

```
$ objdump -Sr hello4
...
08048370 <my_write@plt>:
8048370: ff 25 0c a0 04 08    jmp    *0x804a00c
8048376: 68 00 00 00 00     push  $0x0
804837b: e9 e0 ff ff       jmp    8048360 <.plt>

08048380 <my_strlen@plt>:
8048380: ff 25 10 a0 04 08    jmp    *0x804a010
8048386: 68 08 00 00 00     push  $0x8
804838b: e9 d0 ff ff       jmp    8048360 <.plt>
...
80484aa: 68 a0 85 04 08     push  $0x80485a0
80484af: e8 cc fe ff ff     call  8048380 <my_strlen@plt>
```

- 0x804a00c and 0x804a010 initially point to next instruction
  - Calls dlfixup with relocation index
  - dlfixup needs no relocation because jmp takes relative address

38 / 45



## hello4 relocations

```
$ readelf -r hello4
```

Relocation section '.rel.plt' at offset 0x32c contains 2 entries:

Offset	Info	Type	Sym. Value	Sym. Name
0804a00c	00000107	R_386_JUMP_SLOT	00000000	my_write
0804a010	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

39 / 45

## hello4 shared object contents

### mywrite.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)
```

40 / 45

## 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
```

41 / 45

## 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, CFI)**

42 / 45

## 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)

43 / 45

## 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

44 / 45

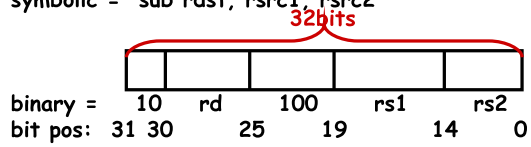


## How?

- Determine binary encoding of desired 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)();
```

```
/* (from glibc sysdeps/unix/sysv/linux/i386/sysdep.h)
   https://sourceware.org/git/?p=glibc.git;a=blob;f=sysdeps/unix/sysv/linux/i386/sysdep
   .h

   Linux takes system call arguments in registers:

       syscall number  %eax          call-clobbered
       arg 1           %ebx          call-saved
       arg 2           %ecx          call-clobbered
       arg 3           %edx          call-clobbered
       arg 4           %esi          call-saved
       arg 5           %edi          call-saved
       arg 6           %ebp          call-saved
*/

#include <sys/syscall.h>

typedef unsigned long size_t;

int my_write(int, const void *, size_t);

int my_errno;

size_t
my_strlen(const char *p)
{
    size_t ret;
    for (ret = 0; p[ret]; ++ret)
        ;
    return ret;
}

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;
}

const char greeting[] = "hello world\n";
int
main(int argc, char **argv, char **envp)
{
    my_write (1, greeting, my_strlen(greeting));
}

void
__libc_start_main(int (*mainp)(int, char **, char **),
                  int argc, char **argv)
{
    mainp(argc, argv, argv + argc + 1);
    asm volatile ("int $0x80" :: "a" (SYS_exit), "b" (0));
}
```

```
#include <sys/syscall.h>

typedef unsigned long size_t;

int my_write(int, const void *, size_t);

static size_t
my_strlen(const char *p)
{
    size_t ret;
    for (ret = 0; p[ret]; ++ret)
        ;
    return ret;
}

const char greeting[] = "hello world\n";
int
main(int argc, char **argv, char **envp)
{
    my_write (1, greeting, my_strlen(greeting));
}

void
__libc_start_main(int (*mainp)(int, char **, char **),
                  int argc, char **argv)
{
    mainp(argc, argv, argv + argc + 1);
    asm volatile ("int $0x80" ::: "a" (SYS_exit), "b" (0));
}
```

```
#include <dlfcn.h>
#include <sys/syscall.h>

const char greeting[] = "hello world\n";
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;

    my_write (1, greeting, my_strlen(greeting));
    return 0;
}

void
__libc_start_main(int (*mainp)(int, char **, char **),
                  int argc, char **argv)
{
    mainp(argc, argv, argv + argc + 1);
    asm volatile ("int $0x80" ::: "a" (SYS_exit), "b" (0));
}
```

```
#include <sys/syscall.h>

typedef unsigned long size_t;

int my_write(int, const void *, size_t);
size_t my_strlen(const char *p);

const char greeting[] = "hello world\n";
int
main(int argc, char **argv, char **envp)
{
    my_write (1, greeting, my_strlen(greeting));
}

void
__libc_start_main(int (*mainp)(int, char **, char **),
                  int argc, char **argv)
{
    mainp(argc, argv, argv + argc + 1);
    asm volatile ("int $0x80" ::: "a" (SYS_exit), "b" (0));
}
```

```
typedef unsigned long size_t;
```

```
size_t
```

```
my_strlen(const char *p)
```

```
{
```

```
    size_t ret;
```

```
    for (ret = 0; p[ret]; ++ret)
```

```
        ;
```

```
    return ret;
```

```
}
```

```
#include <sys/syscall.h>

typedef unsigned long size_t;

int my_errno;

int
my_write(int fd, const void *buf, size_t len)
{
    int ret;
    asm volatile ("pushl %%ebx\n"          // older gcc before version 5
                  "\tmovl %2,%%ebx\n"    // won't allow direct use of
                  "\tint $0x80\n"        // %ebx in PIC code
                  "\tpopl %%ebx"
                  : "=a" (ret)
                  : "0" (SYS_write), "g" (fd), "c" (buf), "d" (len) : "memory");
    if (ret < 0) {
        my_errno = -ret;
        return -1;
    }
    return ret;
}
```



```
.file "hello1.c"
.text
.Ltext0:
.comm my_errno,4,4
.globl my_strlen
.type my_strlen, @function
my_strlen:
.LFB0:
.file 1 "hello1.c"
.loc 1 25 0
.cfi_startproc
pushl %ebp
.cfi_def_cfa_offset 8
.cfi_offset 5, -8
movl %esp, %ebp
.cfi_def_cfa_register 5
subl $16, %esp
.loc 1 27 0
movl $0, -4(%ebp)
jmp .L2
.L3:
.loc 1 27 0 is_stmt 0 discriminator 3
addl $1, -4(%ebp)
.L2:
.loc 1 27 0 discriminator 1
movl 8(%ebp), %edx
movl -4(%ebp), %eax
addl %edx, %eax
movzbl (%eax), %eax
testb %al, %al
jne .L3
.loc 1 29 0 is_stmt 1
movl -4(%ebp), %eax
.loc 1 30 0
leave
.cfi_restore 5
.cfi_def_cfa 4, 4
ret
.cfi_endproc
.LFE0:
.size my_strlen, .-my_strlen
.globl my_write
.type my_write, @function
my_write:
.LFB1:
.loc 1 34 0
.cfi_startproc
pushl %ebp
.cfi_def_cfa_offset 8
.cfi_offset 5, -8
movl %esp, %ebp
.cfi_def_cfa_register 5
pushl %ebx
subl $16, %esp
.cfi_offset 3, -12
.loc 1 36 0
movl $4, %eax
movl 8(%ebp), %ebx
movl 12(%ebp), %ecx
movl 16(%ebp), %edx
#APP
# 36 "hello1.c" 1
.int $0x80
# 0 "" 2
#NO_APP
```

```
    movl    %eax, -8(%ebp)
    .loc 1 38 0
    cmpl    $0, -8(%ebp)
    jns     .L6
    .loc 1 39 0
    movl    -8(%ebp), %eax
    negl    %eax
    movl    %eax, my_errno
    .loc 1 40 0
    movl    $-1, %eax
    jmp     .L7
.L6:
    .loc 1 42 0
    movl    -8(%ebp), %eax
.L7:
    .loc 1 43 0
    addl    $16, %esp
    popl    %ebx
    .cfi_restore 3
    popl    %ebp
    .cfi_restore 5
    .cfi_def_cfa 4, 4
    ret
    .cfi_endproc
.LFE1:
    .size   my_write, .-my_write
    .globl greeting
    .section      .rodata
    .align 4
    .type   greeting, @object
    .size   greeting, 13
greeting:
    .string "hello world\n"
    .text
    .globl main
    .type   main, @function
main:
.LFB2:
    .loc 1 48 0
    .cfi_startproc
    pushl   %ebp
    .cfi_def_cfa_offset 8
    .cfi_offset 5, -8
    movl    %esp, %ebp
    .cfi_def_cfa_register 5
    .loc 1 49 0
    pushl   $greeting
    call    my_strlen
    addl    $4, %esp
    pushl   %eax
    pushl   $greeting
    pushl   $1
    call    my_write
    addl    $12, %esp
    movl    $0, %eax
    .loc 1 50 0
    leave
    .cfi_restore 5
    .cfi_def_cfa 4, 4
    ret
    .cfi_endproc
.LFE2:
    .size   main, .-main
    .globl __libc_start_main
    .type   __libc_start_main, @function
```

```
__libc_start_main:
.LFB3:
    .loc 1 55 0
    .cfi_startproc
    pushl   %ebp
    .cfi_def_cfa_offset 8
    .cfi_offset 5, -8
    movl   %esp, %ebp
    .cfi_def_cfa_register 5
    pushl   %ebx
    subl   $4, %esp
    .cfi_offset 3, -12
    .loc 1 56 0
    movl   12(%ebp), %eax
    addl   $1, %eax
    leal   0(,%eax,4), %edx
    movl   16(%ebp), %eax
    addl   %edx, %eax
    subl   $4, %esp
    pushl   %eax
    pushl   16(%ebp)
    pushl   12(%ebp)
    movl   8(%ebp), %eax
    call   *%eax

.LVL0:
    addl   $16, %esp
    .loc 1 57 0
    movl   $1, %eax
    movl   $0, %edx
    movl   %edx, %ebx

#APP
# 57 "hello1.c" 1
    int $0x80
# 0 "" 2
    .loc 1 58 0
#NO_APP
    nop
    movl   -4(%ebp), %ebx
    leave
    .cfi_restore 5
    .cfi_restore 3
    .cfi_def_cfa 4, 4
    ret
    .cfi_endproc

.LFE3:
    .size   __libc_start_main, .-__libc_start_main

.Letext0:
    .section      .debug_info,"",@progbits
.Ldebug_info0:
    .long   0x1c6
    .value  0x4
    .long   .Ldebug_abbrev0
    .byte   0x4
    .uleb128 0x1
    .long   .LASF12
    .byte   0xc
    .long   .LASF13
    .long   .LASF14
    .long   .Ltext0
    .long   .Letext0-.Ltext0
    .long   .Ldebug_line0
    .uleb128 0x2
    .long   .LASF15
    .byte   0x1
    .byte   0x11
```

```
.long 0x30
.uleb128 0x3
.byte 0x4
.byte 0x7
.long .LASF0
.uleb128 0x4
.long .LASF3
.byte 0x1
.byte 0x15
.long 0x48
.uleb128 0x5
.byte 0x3
.long my_errno
.uleb128 0x5
.byte 0x4
.byte 0x5
.string "int"
.uleb128 0x6
.long 0x72
.long 0x5f
.uleb128 0x7
.long 0x64
.byte 0xc
.byte 0
.uleb128 0x8
.long 0x4f
.uleb128 0x3
.byte 0x4
.byte 0x7
.long .LASF1
.uleb128 0x3
.byte 0x1
.byte 0x6
.long .LASF2
.uleb128 0x8
.long 0x6b
.uleb128 0x4
.long .LASF4
.byte 0x1
.byte 0x2d
.long 0x5f
.uleb128 0x5
.byte 0x3
.long greeting
.uleb128 0x9
.long .LASF16
.byte 0x1
.byte 0x35
.long .LFB3
.long .LFE3-.LFB3
.uleb128 0x1
.byte 0x9c
.long 0xc8
.uleb128 0xa
.long .LASF5
.byte 0x1
.byte 0x35
.long 0xed
.uleb128 0x2
.byte 0x91
.sleb128 0
.uleb128 0xa
.long .LASF6
.byte 0x1
.byte 0x36
```

```
.long 0x48
.uleb128 0x2
.byte 0x91
.sleb128 4
.uleb128 0xa
.long .LASF7
.byte 0x1
.byte 0x36
.long 0xe1
.uleb128 0x2
.byte 0x91
.sleb128 8
.byte 0
.uleb128 0xb
.long 0x48
.long 0xe1
.uleb128 0xc
.long 0x48
.uleb128 0xc
.long 0xe1
.uleb128 0xc
.long 0xe1
.byte 0
.uleb128 0xd
.byte 0x4
.long 0xe7
.uleb128 0xd
.byte 0x4
.long 0x6b
.uleb128 0xd
.byte 0x4
.long 0xc8
.uleb128 0xe
.long .LASF9
.byte 0x1
.byte 0x2f
.long 0x48
.long .LFB2
.long .LFE2-.LFB2
.uleb128 0x1
.byte 0x9c
.long 0x137
.uleb128 0xa
.long .LASF6
.byte 0x1
.byte 0x2f
.long 0x48
.uleb128 0x2
.byte 0x91
.sleb128 0
.uleb128 0xa
.long .LASF7
.byte 0x1
.byte 0x2f
.long 0xe1
.uleb128 0x2
.byte 0x91
.sleb128 4
.uleb128 0xa
.long .LASF8
.byte 0x1
.byte 0x2f
.long 0xe1
.uleb128 0x2
.byte 0x91
```

```
.sleb128 8
.byte 0
.uleb128 0xf
.long .LASF10
.byte 0x1
.byte 0x21
.long 0x48
.long .LFB1
.long .LFE1-.LFB1
.uleb128 0x1
.byte 0x9c
.long 0x188
.uleb128 0x10
.string "fd"
.byte 0x1
.byte 0x21
.long 0x48
.uleb128 0x2
.byte 0x91
.sleb128 0
.uleb128 0x10
.string "buf"
.byte 0x1
.byte 0x21
.long 0x188
.uleb128 0x2
.byte 0x91
.sleb128 4
.uleb128 0x10
.string "len"
.byte 0x1
.byte 0x21
.long 0x25
.uleb128 0x2
.byte 0x91
.sleb128 8
.uleb128 0x11
.string "ret"
.byte 0x1
.byte 0x23
.long 0x48
.uleb128 0x2
.byte 0x91
.sleb128 -16
.byte 0
.uleb128 0xd
.byte 0x4
.long 0x18e
.uleb128 0x12
.uleb128 0xf
.long .LASF11
.byte 0x1
.byte 0x18
.long 0x25
.long .LFB0
.long .LFE0-.LFB0
.uleb128 0x1
.byte 0x9c
.long 0x1c3
.uleb128 0x10
.string "p"
.byte 0x1
.byte 0x18
.long 0x1c3
.uleb128 0x2
```

```
.byte 0x91
.sleb128 0
.uleb128 0x11
.string "ret"
.byte 0x1
.byte 0x1a
.long 0x25
.uleb128 0x2
.byte 0x91
.sleb128 -12
.byte 0
.uleb128 0xd
.byte 0x4
.long 0x72
.byte 0
.section .debug_abbrev,"",@progbits
.Ldebug_abbrev0:
.uleb128 0x1
.uleb128 0x11
.byte 0x1
.uleb128 0x25
.uleb128 0xe
.uleb128 0x13
.uleb128 0xb
.uleb128 0x3
.uleb128 0xe
.uleb128 0x1b
.uleb128 0xe
.uleb128 0x11
.uleb128 0x1
.uleb128 0x12
.uleb128 0x6
.uleb128 0x10
.uleb128 0x17
.byte 0
.byte 0
.uleb128 0x2
.uleb128 0x16
.byte 0
.uleb128 0x3
.uleb128 0xe
.uleb128 0x3a
.uleb128 0xb
.uleb128 0x3b
.uleb128 0xb
.uleb128 0x49
.uleb128 0x13
.byte 0
.byte 0
.uleb128 0x3
.uleb128 0x24
.byte 0
.uleb128 0xb
.uleb128 0xb
.uleb128 0x3e
.uleb128 0xb
.uleb128 0x3
.uleb128 0xe
.byte 0
.byte 0
.uleb128 0x4
.uleb128 0x34
.byte 0
.uleb128 0x3
.uleb128 0xe
```



```
.uleb128 0x3a
.uleb128 0xb
.uleb128 0x3b
.uleb128 0xb
.uleb128 0x49
.uleb128 0x13
.uleb128 0x3f
.uleb128 0x19
.uleb128 0x2
.uleb128 0x18
.byte 0
.byte 0
.uleb128 0x5
.uleb128 0x24
.byte 0
.uleb128 0xb
.uleb128 0xb
.uleb128 0x3e
.uleb128 0xb
.uleb128 0x3
.uleb128 0x8
.byte 0
.byte 0
.uleb128 0x6
.uleb128 0x1
.byte 0x1
.uleb128 0x49
.uleb128 0x13
.uleb128 0x1
.uleb128 0x13
.byte 0
.byte 0
.uleb128 0x7
.uleb128 0x21
.byte 0
.uleb128 0x49
.uleb128 0x13
.uleb128 0x2f
.uleb128 0xb
.byte 0
.byte 0
.uleb128 0x8
.uleb128 0x26
.byte 0
.uleb128 0x49
.uleb128 0x13
.byte 0
.byte 0
.uleb128 0x9
.uleb128 0x2e
.byte 0x1
.uleb128 0x3f
.uleb128 0x19
.uleb128 0x3
.uleb128 0xe
.uleb128 0x3a
.uleb128 0xb
.uleb128 0x3b
.uleb128 0xb
.uleb128 0x27
.uleb128 0x19
.uleb128 0x11
.uleb128 0x1
.uleb128 0x12
.uleb128 0x6
```

```
.uleb128 0x40
.uleb128 0x18
.uleb128 0x2116
.uleb128 0x19
.uleb128 0x1
.uleb128 0x13
.byte 0
.byte 0
.uleb128 0xa
.uleb128 0x5
.byte 0
.uleb128 0x3
.uleb128 0xe
.uleb128 0x3a
.uleb128 0xb
.uleb128 0x3b
.uleb128 0xb
.uleb128 0x49
.uleb128 0x13
.uleb128 0x2
.uleb128 0x18
.byte 0
.byte 0
.uleb128 0xb
.uleb128 0x15
.byte 0x1
.uleb128 0x27
.uleb128 0x19
.uleb128 0x49
.uleb128 0x13
.uleb128 0x1
.uleb128 0x13
.byte 0
.byte 0
.uleb128 0xc
.uleb128 0x5
.byte 0
.uleb128 0x49
.uleb128 0x13
.byte 0
.byte 0
.uleb128 0xd
.uleb128 0xf
.byte 0
.uleb128 0xb
.uleb128 0xb
.uleb128 0x49
.uleb128 0x13
.byte 0
.byte 0
.uleb128 0xe
.uleb128 0x2e
.byte 0x1
.uleb128 0x3f
.uleb128 0x19
.uleb128 0x3
.uleb128 0xe
.uleb128 0x3a
.uleb128 0xb
.uleb128 0x3b
.uleb128 0xb
.uleb128 0x27
.uleb128 0x19
.uleb128 0x49
.uleb128 0x13
```

```
.uleb128 0x11
.uleb128 0x1
.uleb128 0x12
.uleb128 0x6
.uleb128 0x40
.uleb128 0x18
.uleb128 0x2116
.uleb128 0x19
.uleb128 0x1
.uleb128 0x13
.byte 0
.byte 0
.uleb128 0xf
.uleb128 0x2e
.byte 0x1
.uleb128 0x3f
.uleb128 0x19
.uleb128 0x3
.uleb128 0xe
.uleb128 0x3a
.uleb128 0xb
.uleb128 0x3b
.uleb128 0xb
.uleb128 0x27
.uleb128 0x19
.uleb128 0x49
.uleb128 0x13
.uleb128 0x11
.uleb128 0x1
.uleb128 0x12
.uleb128 0x6
.uleb128 0x40
.uleb128 0x18
.uleb128 0x2117
.uleb128 0x19
.uleb128 0x1
.uleb128 0x13
.byte 0
.byte 0
.uleb128 0x10
.uleb128 0x5
.byte 0
.uleb128 0x3
.uleb128 0x8
.uleb128 0x3a
.uleb128 0xb
.uleb128 0x3b
.uleb128 0xb
.uleb128 0x49
.uleb128 0x13
.uleb128 0x2
.uleb128 0x18
.byte 0
.byte 0
.uleb128 0x11
.uleb128 0x34
.byte 0
.uleb128 0x3
.uleb128 0x8
.uleb128 0x3a
.uleb128 0xb
.uleb128 0x3b
.uleb128 0xb
.uleb128 0x49
.uleb128 0x13
```

```
.uleb128 0x2
.uleb128 0x18
.byte 0
.byte 0
.uleb128 0x12
.uleb128 0x26
.byte 0
.byte 0
.byte 0
.byte 0
.section          .debug_aranges,"",@progbits
.long 0x1c
.value 0x2
.long .Ldebug_info0
.byte 0x4
.byte 0
.value 0
.value 0
.long .Ltext0
.long .Letext0-.Ltext0
.long 0
.long 0
.section          .debug_line,"",@progbits
.Ldebug_line0:
.section          .debug_str,"MS",@progbits,1
.LASF15:
.string "size_t"
.LASF14:
.string "/home/dm/class/18wi-cs140/notes/linking/hello"
.LASF9:
.string "main"
.LASF3:
.string "my_errno"
.LASF0:
.string "long unsigned int"
.LASF10:
.string "my_write"
.LASF4:
.string "greeting"
.LASF8:
.string "envp"
.LASF2:
.string "char"
.LASF12:
.string "GNU C11 7.3.0 -m32 -mtune=generic -march=x86-64 -g -O0 -fno-pic -fno-b
uilty"
.LASF11:
.string "my_strlen"
.LASF6:
.string "argc"
.LASF5:
.string "mainp"
.LASF7:
.string "argv"
.LASF1:
.string "unsigned int"
.LASF16:
.string "__libc_start_main"
.LASF13:
.string "hello1.c"
.ident "GCC: (GNU) 7.3.0"
.section          .note.GNU-stack,"",@progbits
```

```
.file "hello4.c"
.text
.Ltext0:
.globl greeting
.section .rodata
.align 4
.type greeting, @object
.size greeting, 13
greeting:
.string "hello world\n"
.text
.globl main
.type main, @function
main:
.LFB0:
.file 1 "hello4.c"
.loc 1 11 0
.cfi_startproc
leal 4(%esp), %ecx
.cfi_def_cfa 1, 0
andl $-16, %esp
pushl -4(%ecx)
pushl %ebp
.cfi_escape 0x10,0x5,0x2,0x75,0
movl %esp, %ebp
pushl %ecx
.cfi_escape 0xf,0x3,0x75,0x7c,0x6
subl $4, %esp
.loc 1 12 0
subl $12, %esp
pushl $greeting
call my_strlen
addl $16, %esp
subl $4, %esp
pushl %eax
pushl $greeting
pushl $1
call my_write
addl $16, %esp
movl $0, %eax
.loc 1 13 0
movl -4(%ebp), %ecx
.cfi_def_cfa 1, 0
leave
.cfi_restore 5
leal -4(%ecx), %esp
.cfi_def_cfa 4, 4
ret
.cfi_endproc
.LFE0:
.size main, .-main
.globl __libc_start_main
.type __libc_start_main, @function
__libc_start_main:
.LFB1:
.loc 1 18 0
.cfi_startproc
pushl %ebp
.cfi_def_cfa_offset 8
.cfi_offset 5, -8
movl %esp, %ebp
.cfi_def_cfa_register 5
pushl %ebx
subl $4, %esp
.cfi_offset 3, -12
```

```
.loc 1 19 0
movl 12(%ebp), %eax
addl $1, %eax
leal 0(,%eax,4), %edx
movl 16(%ebp), %eax
addl %edx, %eax
subl $4, %esp
pushl %eax
pushl 16(%ebp)
pushl 12(%ebp)
movl 8(%ebp), %eax
call *%eax
.LVL0:
addl $16, %esp
.loc 1 20 0
movl $1, %eax
movl $0, %edx
movl %edx, %ebx
#APP
# 20 "hello4.c" 1
int $0x80
# 0 "" 2
.loc 1 21 0
#NO_APP
nop
movl -4(%ebp), %ebx
leave
.cfi_restore 5
.cfi_restore 3
.cfi_def_cfa 4, 4
ret
.cfi_endproc
.LFE1:
.size __libc_start_main, .-__libc_start_main
.Letext0:
.section .debug_info,"",@progbits
.Ldebug_info0:
.long 0x114
.value 0x4
.long .Ldebug_abbrev0
.byte 0x4
.uleb128 0x1
.long .LASF9
.byte 0xc
.long .LASF10
.long .LASF11
.long .Ltext0
.long .Letext0-.Ltext0
.long .Ldebug_line0
.uleb128 0x2
.byte 0x4
.byte 0x7
.long .LASF0
.uleb128 0x3
.long 0x4f
.long 0x3c
.uleb128 0x4
.long 0x41
.byte 0xc
.byte 0
.uleb128 0x5
.long 0x2c
.uleb128 0x2
.byte 0x4
.byte 0x7
```

```
.long    .LASF1
.uleb128 0x2
.byte    0x1
.byte    0x6
.long    .LASF2
.uleb128 0x5
.long    0x48
.uleb128 0x6
.long    .LASF12
.byte    0x1
.byte    0x8
.long    0x3c
.uleb128 0x5
.byte    0x3
.long    greeting
.uleb128 0x7
.long    .LASF6
.byte    0x1
.byte    0x10
.long    .LFB1
.long    .LFE1-.LFB1
.uleb128 0x1
.byte    0x9c
.long    0xa5
.uleb128 0x8
.long    .LASF3
.byte    0x1
.byte    0x10
.long    0xd1
.uleb128 0x2
.byte    0x91
.sleb128 0
.uleb128 0x8
.long    .LASF4
.byte    0x1
.byte    0x11
.long    0xbe
.uleb128 0x2
.byte    0x91
.sleb128 4
.uleb128 0x8
.long    .LASF5
.byte    0x1
.byte    0x11
.long    0xc5
.uleb128 0x2
.byte    0x91
.sleb128 8
.byte    0
.uleb128 0x9
.long    0xbe
.long    0xbe
.uleb128 0xa
.long    0xbe
.uleb128 0xa
.long    0xc5
.uleb128 0xa
.long    0xc5
.byte    0
.uleb128 0xb
.byte    0x4
.byte    0x5
.string  "int"
.uleb128 0xc
.byte    0x4
```



```
.long 0xcb
.uleb128 0xc
.byte 0x4
.long 0x48
.uleb128 0xc
.byte 0x4
.long 0xa5
.uleb128 0xd
.long .LASF7
.byte 0x1
.byte 0xa
.long 0xbe
.long .LFB0
.long .LFE0-.LFB0
.uleb128 0x1
.byte 0x9c
.uleb128 0x8
.long .LASF4
.byte 0x1
.byte 0xa
.long 0xbe
.uleb128 0x2
.byte 0x91
.sleb128 0
.uleb128 0x8
.long .LASF5
.byte 0x1
.byte 0xa
.long 0xc5
.uleb128 0x2
.byte 0x91
.sleb128 4
.uleb128 0x8
.long .LASF8
.byte 0x1
.byte 0xa
.long 0xc5
.uleb128 0x2
.byte 0x91
.sleb128 8
.byte 0
.byte 0
.section .debug_abbrev,"",@progbits
.Ldebug_abbrev0:
.uleb128 0x1
.uleb128 0x11
.byte 0x1
.uleb128 0x25
.uleb128 0xe
.uleb128 0x13
.uleb128 0xb
.uleb128 0x3
.uleb128 0xe
.uleb128 0x1b
.uleb128 0xe
.uleb128 0x11
.uleb128 0x1
.uleb128 0x12
.uleb128 0x6
.uleb128 0x10
.uleb128 0x17
.byte 0
.byte 0
.uleb128 0x2
.uleb128 0x24
```

```
.byte 0
.uleb128 0xb
.uleb128 0xb
.uleb128 0x3e
.uleb128 0xb
.uleb128 0x3
.uleb128 0xe
.byte 0
.byte 0
.uleb128 0x3
.uleb128 0x1
.byte 0x1
.uleb128 0x49
.uleb128 0x13
.uleb128 0x1
.uleb128 0x13
.byte 0
.byte 0
.uleb128 0x4
.uleb128 0x21
.byte 0
.uleb128 0x49
.uleb128 0x13
.uleb128 0x2f
.uleb128 0xb
.byte 0
.byte 0
.uleb128 0x5
.uleb128 0x26
.byte 0
.uleb128 0x49
.uleb128 0x13
.byte 0
.byte 0
.uleb128 0x6
.uleb128 0x34
.byte 0
.uleb128 0x3
.uleb128 0xe
.uleb128 0x3a
.uleb128 0xb
.uleb128 0x3b
.uleb128 0xb
.uleb128 0x49
.uleb128 0x13
.uleb128 0x3f
.uleb128 0x19
.uleb128 0x2
.uleb128 0x18
.byte 0
.byte 0
.uleb128 0x7
.uleb128 0x2e
.byte 0x1
.uleb128 0x3f
.uleb128 0x19
.uleb128 0x3
.uleb128 0xe
.uleb128 0x3a
.uleb128 0xb
.uleb128 0x3b
.uleb128 0xb
.uleb128 0x27
.uleb128 0x19
.uleb128 0x11
```

```
.uleb128 0x1
.uleb128 0x12
.uleb128 0x6
.uleb128 0x40
.uleb128 0x18
.uleb128 0x2116
.uleb128 0x19
.uleb128 0x1
.uleb128 0x13
.byte 0
.byte 0
.uleb128 0x8
.uleb128 0x5
.byte 0
.uleb128 0x3
.uleb128 0xe
.uleb128 0x3a
.uleb128 0xb
.uleb128 0x3b
.uleb128 0xb
.uleb128 0x49
.uleb128 0x13
.uleb128 0x2
.uleb128 0x18
.byte 0
.byte 0
.uleb128 0x9
.uleb128 0x15
.byte 0x1
.uleb128 0x27
.uleb128 0x19
.uleb128 0x49
.uleb128 0x13
.uleb128 0x1
.uleb128 0x13
.byte 0
.byte 0
.uleb128 0xa
.uleb128 0x5
.byte 0
.uleb128 0x49
.uleb128 0x13
.byte 0
.byte 0
.uleb128 0xb
.uleb128 0x24
.byte 0
.uleb128 0xb
.uleb128 0xb
.uleb128 0x3e
.uleb128 0xb
.uleb128 0x3
.uleb128 0x8
.byte 0
.byte 0
.uleb128 0xc
.uleb128 0xf
.byte 0
.uleb128 0xb
.uleb128 0xb
.uleb128 0x49
.uleb128 0x13
.byte 0
.byte 0
.uleb128 0xd
```

```
.uleb128 0x2e
.byte 0x1
.uleb128 0x3f
.uleb128 0x19
.uleb128 0x3
.uleb128 0xe
.uleb128 0x3a
.uleb128 0xb
.uleb128 0x3b
.uleb128 0xb
.uleb128 0x27
.uleb128 0x19
.uleb128 0x49
.uleb128 0x13
.uleb128 0x11
.uleb128 0x1
.uleb128 0x12
.uleb128 0x6
.uleb128 0x40
.uleb128 0x18
.uleb128 0x2116
.uleb128 0x19
.byte 0
.byte 0
.byte 0
.section          .debug_aranges,"",@progbits
.long 0x1c
.value 0x2
.long .Ldebug_info0
.byte 0x4
.byte 0
.value 0
.value 0
.long .Ltext0
.long .Letext0-.Ltext0
.long 0
.long 0
.section          .debug_line,"",@progbits
.Ldebug_line0:
.section          .debug_str,"MS",@progbits,1
.LASF9:
.string "GNU C11 7.3.0 -m32 -mtune=generic -march=x86-64 -g -O0 -fno-pic -fno-b
uiltin"
.LASF1:
.string "unsigned int"
.LASF0:
.string "long unsigned int"
.LASF8:
.string "envp"
.LASF3:
.string "mainp"
.LASF12:
.string "greeting"
.LASF11:
.string "/home/dm/class/18wi-cs140/notes/linking/hello"
.LASF4:
.string "argc"
.LASF10:
.string "hello4.c"
.LASF7:
.string "main"
.LASF2:
.string "char"
.LASF6:
.string "__libc_start_main"
```

.LASF5:

```
.string "argv"  
.ident "GCC: (GNU) 7.3.0"  
.section .note.GNU-stack,"",@progbits
```

```
.file "mywrite.c"
.text
.Ltext0:
.comm my_errno,4,4
.globl my_write
.type my_write, @function
my_write:
.LFB0:
.file 1 "mywrite.c"
.loc 1 9 0
.cfi_startproc
pushl %ebp
.cfi_def_cfa_offset 8
.cfi_offset 5, -8
movl %esp, %ebp
.cfi_def_cfa_register 5
subl $16, %esp
.loc 1 11 0
movl $4, %eax
movl 12(%ebp), %ecx
movl 16(%ebp), %edx
#APP
# 11 "mywrite.c" 1
pushl %ebx
movl 8(%ebp), %ebx
int $0x80
popl %ebx
# 0 "" 2
#NO_APP
movl %eax, -4(%ebp)
.loc 1 17 0
cmpl $0, -4(%ebp)
jns .L2
.loc 1 18 0
movl -4(%ebp), %eax
negl %eax
movl %eax, my_errno
.loc 1 19 0
movl $-1, %eax
jmp .L3
.L2:
.loc 1 21 0
movl -4(%ebp), %eax
.L3:
.loc 1 22 0
leave
.cfi_restore 5
.cfi_def_cfa 4, 4
ret
.cfi_endproc
.LFE0:
.size my_write, .-my_write
.Letext0:
.section .debug_info,"",@progbits
.Ldebug_info0:
.long 0xa4
.value 0x4
.long .Ldebug_abbrev0
.byte 0x4
.uleb128 0x1
.long .LASF1
.byte 0xc
.long .LASF2
.long .LASF3
.long .Ltext0
```

```
.long .Letext0-.Ltext0
.long .Ldebug_line0
.uleb128 0x2
.long .LASF4
.byte 0x1
.byte 0x3
.long 0x30
.uleb128 0x3
.byte 0x4
.byte 0x7
.long .LASF0
.uleb128 0x4
.long .LASF5
.byte 0x1
.byte 0x5
.long 0x48
.uleb128 0x5
.byte 0x3
.long my_errno
.uleb128 0x5
.byte 0x4
.byte 0x5
.string "int"
.uleb128 0x6
.long .LASF6
.byte 0x1
.byte 0x8
.long 0x48
.long .LFB0
.long .LFE0-.LFB0
.uleb128 0x1
.byte 0x9c
.long 0xa0
.uleb128 0x7
.string "fd"
.byte 0x1
.byte 0x8
.long 0x48
.uleb128 0x2
.byte 0x91
.sleb128 0
.uleb128 0x7
.string "buf"
.byte 0x1
.byte 0x8
.long 0xa0
.uleb128 0x2
.byte 0x91
.sleb128 4
.uleb128 0x7
.string "len"
.byte 0x1
.byte 0x8
.long 0x25
.uleb128 0x2
.byte 0x91
.sleb128 8
.uleb128 0x8
.string "ret"
.byte 0x1
.byte 0xa
.long 0x48
.uleb128 0x2
.byte 0x91
.sleb128 -12
```



```
.byte 0
.uleb128 0x9
.byte 0x4
.long 0xa6
.uleb128 0xa
.byte 0
.section .debug_abbrev,"",@progbits
.Ldebug_abbrev0:
.uleb128 0x1
.uleb128 0x11
.byte 0x1
.uleb128 0x25
.uleb128 0xe
.uleb128 0x13
.uleb128 0xb
.uleb128 0x3
.uleb128 0xe
.uleb128 0x1b
.uleb128 0xe
.uleb128 0x11
.uleb128 0x1
.uleb128 0x12
.uleb128 0x6
.uleb128 0x10
.uleb128 0x17
.byte 0
.byte 0
.uleb128 0x2
.uleb128 0x16
.byte 0
.uleb128 0x3
.uleb128 0xe
.uleb128 0x3a
.uleb128 0xb
.uleb128 0x3b
.uleb128 0xb
.uleb128 0x49
.uleb128 0x13
.byte 0
.byte 0
.uleb128 0x3
.uleb128 0x24
.byte 0
.uleb128 0xb
.uleb128 0xb
.uleb128 0x3e
.uleb128 0xb
.uleb128 0x3
.uleb128 0xe
.byte 0
.byte 0
.uleb128 0x4
.uleb128 0x34
.byte 0
.uleb128 0x3
.uleb128 0xe
.uleb128 0x3a
.uleb128 0xb
.uleb128 0x3b
.uleb128 0xb
.uleb128 0x49
.uleb128 0x13
.uleb128 0x3f
.uleb128 0x19
.uleb128 0x2
```

```
.uleb128 0x18
.byte 0
.byte 0
.uleb128 0x5
.uleb128 0x24
.byte 0
.uleb128 0xb
.uleb128 0xb
.uleb128 0x3e
.uleb128 0xb
.uleb128 0x3
.uleb128 0x8
.byte 0
.byte 0
.uleb128 0x6
.uleb128 0x2e
.byte 0x1
.uleb128 0x3f
.uleb128 0x19
.uleb128 0x3
.uleb128 0xe
.uleb128 0x3a
.uleb128 0xb
.uleb128 0x3b
.uleb128 0xb
.uleb128 0x27
.uleb128 0x19
.uleb128 0x49
.uleb128 0x13
.uleb128 0x11
.uleb128 0x1
.uleb128 0x12
.uleb128 0x6
.uleb128 0x40
.uleb128 0x18
.uleb128 0x2117
.uleb128 0x19
.uleb128 0x1
.uleb128 0x13
.byte 0
.byte 0
.uleb128 0x7
.uleb128 0x5
.byte 0
.uleb128 0x3
.uleb128 0x8
.uleb128 0x3a
.uleb128 0xb
.uleb128 0x3b
.uleb128 0xb
.uleb128 0x49
.uleb128 0x13
.uleb128 0x2
.uleb128 0x18
.byte 0
.byte 0
.uleb128 0x8
.uleb128 0x34
.byte 0
.uleb128 0x3
.uleb128 0x8
.uleb128 0x3a
.uleb128 0xb
.uleb128 0x3b
.uleb128 0xb
```

```
.uleb128 0x49
.uleb128 0x13
.uleb128 0x2
.uleb128 0x18
.byte 0
.byte 0
.uleb128 0x9
.uleb128 0xf
.byte 0
.uleb128 0xb
.uleb128 0xb
.uleb128 0x49
.uleb128 0x13
.byte 0
.byte 0
.uleb128 0xa
.uleb128 0x26
.byte 0
.byte 0
.byte 0
.byte 0
.section .debug_aranges,"",@progbits
.long 0x1c
.value 0x2
.long .Ldebug_info0
.byte 0x4
.byte 0
.value 0
.value 0
.long .Ltext0
.long .Letext0-.Ltext0
.long 0
.long 0
.section .debug_line,"",@progbits
.Ldebug_line0:
.section .debug_str,"MS",@progbits,1
.LASF1:
.string "GNU C11 7.3.0 -m32 -mtune=generic -march=x86-64 -g -O0 -fno-pic -fno-b
uilty"
.LASF0:
.string "long unsigned int"
.LASF4:
.string "size_t"
.LASF3:
.string "/home/dm/class/18wi-cs140/notes/linking/hello"
.LASF2:
.string "mywrite.c"
.LASF5:
.string "my_errno"
.LASF6:
.string "my_write"
.ident "GCC: (GNU) 7.3.0"
.section .note.GNU-stack,"",@progbits
```

```
.file "mywrite.c"
.text
.Ltext0:
.comm my_errno,4,4
.globl my_write
.type my_write, @function
my_write:
.LFB0:
.file 1 "mywrite.c"
.loc 1 9 0
.cfi_startproc
pushl %ebp
.cfi_def_cfa_offset 8
.cfi_offset 5, -8
movl %esp, %ebp
.cfi_def_cfa_register 5
pushl %ebx
subl $16, %esp
.cfi_offset 3, -12
call __x86.get_pc_thunk.bx
addl $_GLOBAL_OFFSET_TABLE_, %ebx
.loc 1 11 0
movl $4, %eax
movl 12(%ebp), %ecx
movl 16(%ebp), %edx
#APP
# 11 "mywrite.c" 1
pushl %ebx
movl 8(%ebp), %ebx
int $0x80
popl %ebx
# 0 "" 2
#NO_APP
movl %eax, -8(%ebp)
.loc 1 17 0
cmpl $0, -8(%ebp)
jns .L2
.loc 1 18 0
movl -8(%ebp), %eax
negl %eax
movl %eax, %edx
movl my_errno@GOT(%ebx), %eax
movl %edx, (%eax)
.loc 1 19 0
movl $-1, %eax
jmp .L3
.L2:
.loc 1 21 0
movl -8(%ebp), %eax
.L3:
.loc 1 22 0
addl $16, %esp
popl %ebx
.cfi_restore 3
popl %ebp
.cfi_restore 5
.cfi_def_cfa 4, 4
ret
.cfi_endproc
.LFE0:
.size my_write, .-my_write
.section .text.__x86.get_pc_thunk.bx,"axG",@progbits,__x86.get_pc_thunk.
bx,comdat
.globl __x86.get_pc_thunk.bx
.hidden __x86.get_pc_thunk.bx
```

```
.type __x86.get_pc_thunk.bx, @function
__x86.get_pc_thunk.bx:
.LFB1:
    .cfi_startproc
    movl    (%esp), %ebx
    ret
    .cfi_endproc
.LFE1:
    .text
.Letext0:
    .section      .debug_info,"",@progbits
.Ldebug_info0:
    .long    0xa4
    .value   0x4
    .long    .Ldebug_abbrev0
    .byte    0x4
    .uleb128 0x1
    .long    .LASF1
    .byte    0xc
    .long    .LASF2
    .long    .LASF3
    .long    .Ltext0
    .long    .Letext0-.Ltext0
    .long    .Ldebug_line0
    .uleb128 0x2
    .long    .LASF4
    .byte    0x1
    .byte    0x3
    .long    0x30
    .uleb128 0x3
    .byte    0x4
    .byte    0x7
    .long    .LASF0
    .uleb128 0x4
    .long    .LASF5
    .byte    0x1
    .byte    0x5
    .long    0x48
    .uleb128 0x5
    .byte    0x3
    .long    my_errno
    .uleb128 0x5
    .byte    0x4
    .byte    0x5
    .string  "int"
    .uleb128 0x6
    .long    .LASF6
    .byte    0x1
    .byte    0x8
    .long    0x48
    .long    .LFB0
    .long    .LFE0-.LFB0
    .uleb128 0x1
    .byte    0x9c
    .long    0xa0
    .uleb128 0x7
    .string  "fd"
    .byte    0x1
    .byte    0x8
    .long    0x48
    .uleb128 0x2
    .byte    0x91
    .sleb128 0
    .uleb128 0x7
    .string  "buf"
```

```
.byte 0x1
.byte 0x8
.long 0xa0
.uleb128 0x2
.byte 0x91
.sleb128 4
.uleb128 0x7
.string "len"
.byte 0x1
.byte 0x8
.long 0x25
.uleb128 0x2
.byte 0x91
.sleb128 8
.uleb128 0x8
.string "ret"
.byte 0x1
.byte 0xa
.long 0x48
.uleb128 0x2
.byte 0x91
.sleb128 -16
.byte 0
.uleb128 0x9
.byte 0x4
.long 0xa6
.uleb128 0xa
.byte 0
.section .debug_abbrev,"",@progbits
.Ldebug_abbrev0:
.uleb128 0x1
.uleb128 0x11
.byte 0x1
.uleb128 0x25
.uleb128 0xe
.uleb128 0x13
.uleb128 0xb
.uleb128 0x3
.uleb128 0xe
.uleb128 0x1b
.uleb128 0xe
.uleb128 0x11
.uleb128 0x1
.uleb128 0x12
.uleb128 0x6
.uleb128 0x10
.uleb128 0x17
.byte 0
.byte 0
.uleb128 0x2
.uleb128 0x16
.byte 0
.uleb128 0x3
.uleb128 0xe
.uleb128 0x3a
.uleb128 0xb
.uleb128 0x3b
.uleb128 0xb
.uleb128 0x49
.uleb128 0x13
.byte 0
.byte 0
.uleb128 0x3
.uleb128 0x24
.byte 0
```

```
.uleb128 0xb
.uleb128 0xb
.uleb128 0x3e
.uleb128 0xb
.uleb128 0x3
.uleb128 0xe
.byte 0
.byte 0
.uleb128 0x4
.uleb128 0x34
.byte 0
.uleb128 0x3
.uleb128 0xe
.uleb128 0x3a
.uleb128 0xb
.uleb128 0x3b
.uleb128 0xb
.uleb128 0x49
.uleb128 0x13
.uleb128 0x3f
.uleb128 0x19
.uleb128 0x2
.uleb128 0x18
.byte 0
.byte 0
.uleb128 0x5
.uleb128 0x24
.byte 0
.uleb128 0xb
.uleb128 0xb
.uleb128 0x3e
.uleb128 0xb
.uleb128 0x3
.uleb128 0x8
.byte 0
.byte 0
.uleb128 0x6
.uleb128 0x2e
.byte 0x1
.uleb128 0x3f
.uleb128 0x19
.uleb128 0x3
.uleb128 0xe
.uleb128 0x3a
.uleb128 0xb
.uleb128 0x3b
.uleb128 0xb
.uleb128 0x27
.uleb128 0x19
.uleb128 0x49
.uleb128 0x13
.uleb128 0x11
.uleb128 0x1
.uleb128 0x12
.uleb128 0x6
.uleb128 0x40
.uleb128 0x18
.uleb128 0x2117
.uleb128 0x19
.uleb128 0x1
.uleb128 0x13
.byte 0
.byte 0
.uleb128 0x7
.uleb128 0x5
```

```
.byte 0
.uleb128 0x3
.uleb128 0x8
.uleb128 0x3a
.uleb128 0xb
.uleb128 0x3b
.uleb128 0xb
.uleb128 0x49
.uleb128 0x13
.uleb128 0x2
.uleb128 0x18
.byte 0
.byte 0
.uleb128 0x8
.uleb128 0x34
.byte 0
.uleb128 0x3
.uleb128 0x8
.uleb128 0x3a
.uleb128 0xb
.uleb128 0x3b
.uleb128 0xb
.uleb128 0x49
.uleb128 0x13
.uleb128 0x2
.uleb128 0x18
.byte 0
.byte 0
.uleb128 0x9
.uleb128 0xf
.byte 0
.uleb128 0xb
.uleb128 0xb
.uleb128 0x49
.uleb128 0x13
.byte 0
.byte 0
.uleb128 0xa
.uleb128 0x26
.byte 0
.byte 0
.byte 0
.byte 0
.section .debug_aranges,"",@progbits
.long 0x1c
.value 0x2
.long .Ldebug_info0
.byte 0x4
.byte 0
.value 0
.value 0
.long .Ltext0
.long .Letext0-.Ltext0
.long 0
.long 0
.section .debug_line,"",@progbits
.Ldebug_line0:
.section .debug_str,"MS",@progbits,1
.LASF0:
.string "long unsigned int"
.LASF4:
.string "size_t"
.LASF3:
.string "/home/dm/class/18wi-cs140/notes/linking/hello"
.LASF1:
```



```
.string "GNU C11 7.3.0 -m32 -mtune=generic -march=x86-64 -g -O0 -fno-builtin -f
PIC"
.LASF2:
.string "mywrite.c"
.LASF5:
.string "my_errno"
.LASF6:
.string "my_write"
.ident "GCC: (GNU) 7.3.0"
.section .note.GNU-stack,"",@progbits
```