CS 33

Linkers

gcc Steps

1) **Compile**
   - **to start here, supply .c file**
   - **to stop here: `gcc -S` (produces .s file)**
   - **(if not stopping here, gcc compiles directly into machine code, bypassing the assembler)**

2) **Assemble**
   - **to start here, supply .s file**
   - **to stop here: `gcc -c` (produces .o file)**

3) **Link**
   - **to start here, supply .o file**

The Linker

- **An executable program is one that is ready to be loaded into memory**
- **The linker (known as ld: /usr/bin/ld) creates such executables from:**
  - object files produced by the compiler/assembler
  - collections of object files (known as libraries or archives)
  - and more we'll get to soon ...

The technology described here is current as of around 1990 and is known as static linking. We discuss static linking first, then move on to dynamic linking, which is commonplace today.

Processes and Address Spaces

- **The *process***
  - **OS entity representing a computation**
    » **sort of a virtual computer**
    » **encapsulates**
      - **processors (threads)**
      - **storage**
        - **containing code and data**
          - **address space**
          - **open files**
      - **identity**

The Unix Address Space

stack
↓ ↑
dynamic
bss
data
text
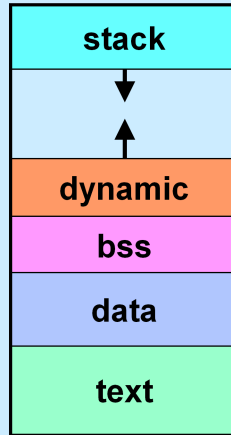
A Unix process's address space appears to be three regions of memory: a read-only *text* region (containing executable code); a read-write region consisting of initialized *data* (simply called data), uninitialized data (*BSS* — a directive from an ancient assembler (for the IBM 704 series of computers), standing for Block Started by Symbol and used to reserve space for uninitialized storage), and a *dynamic area*; and a second read-write region containing the process's user *stack* (a standard Unix process contains only one thread of control).

The first area of read-write storage is often collectively called the data region. Its dynamic portion grows in response to *sbrk* system calls. Most programmers do not use this system call directly, but instead use the *malloc* and *free* library routines, which manage the dynamic area and allocate memory when needed by in turn executing *sbrk* system calls.

The stack region grows implicitly: whenever an attempt is made to reference beyond the current end of stack, the stack is implicitly grown to the new reference. (There are system-wide and per-process limits on the maximum data and stack sizes of processes.)

Note that here we are discussing strictly single-threaded processes. Later we will discuss multi-threaded processes, whose address spaces contain multiple stacks.

Linker's Job

- **Piece together components of program**
  - **arrange within address space**
    - » **code (and read-only data) goes into text region**
    - » **initialized data goes into data region**
    - » **uninitialized data goes into bss region**
- **Modify address references, as necessary**

A Program

```
int nprimes = 100;                                         data
int *prime, *prime2;                                       bss
int main() {
    int i, j, current = 1;
    prime = (int *)malloc(nprimes*sizeof(*prime));         dynamic
    prime2 = (int *)malloc(nprimes*sizeof(*prime2));
    prime[0] = 2; prime2[0] = 2*2;
    for (i=1; i<nprimes; i++) {
    NewCandidate:
        current += 2;
        for (j=0; prime2[j] <= current; j++) {
            if (current % prime[j] == 0)
                goto NewCandidate;
        }
        prime[i] = current; prime2[i] = current*current;
    }
    return 0;
}
```

text

The code is an implementation of the "sieve of Eratosthenes", an early (~200 BC) algorithm for enumerating prime numbers.

The *malloc* routine allocates storage within the dynamic region. We discuss it in detail in an upcoming lecture.

... with Output

```
int nprimes = 100;
int *prime, *prime2;
int main() {
    ...
    printcol(5);
    return 0;
}


void printcol(int ncols) {
    int i, j;
    int nrows = (nprimes+ncols-1)/ncols;
    for (i = 0; i<nrows; i++) {
        for (j=0; (j<ncols) && (i+nrows*j < nvals); j++) {
            printf("%6d", prime[i + nrows*j]);
        }
        printf("\n");
    }
}
```

... Compiled Separately

should refer to same thing

```
int nprimes = 100;
int *prime, *prime2;
int main() {
    ...
    printcol(5);
    return 0;
}
```

ditto

primes.c

```
extern int nprimes;
int *prime;
void printcol(int ncols) {
    int i, j;
    int nrows = (nprimes+ncols-1)/ncols;
    for (i = 0; i<nrows; i++) {
        for (j=0; (j<ncols)
               && (i+nrows*j < nvals); j++) {
            printf("%6d", prime[i + nrows*j]);
        }
        printf("\n");
    }
}
```
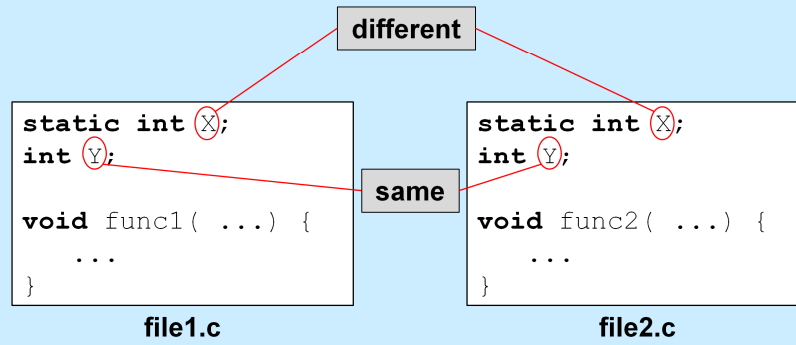
printcol.c

gcc –c primes.c
gcc –c printcol.c
gcc –o primes primes.o printcol.o

In the first two invocations of gcc, the "-c" flag tells it to compile the C code and produce an object (".o") file, but not to go any further (and thus not to produce an executable program). In the third invocation, gcc invokes the ld (linker) program to combine the two object files into an executable program. As we discuss soon, it will also bring in code (such as printf) from libraries.

Global Variables

- **Initialized vs. uninitialized**
  - **initialized allocated in *data* section**
  - **uninitialized allocated in *bss* section**
    - » **implicitly initialized to zero**
- **File scope vs. program scope**
  - **static global variables known only within file that declares them**
    - » **two of same name in different files are different**
    - » **e.g., `static int X;`**
  - **non-static global variables potentially shared across all files**
    - » **two of same name in different files are same**
    - » **e.g., `int X;`**

Scope


```
            different

static int X;        static int X;
int Y;    same       int Y;

void func1( ...) {    void func2( ...) {
    ...                  ...
}                    }
      file1.c               file2.c
```

Static Local Variables

```
int *sub1() {                 int *sub2() {
   int var;                       static int var;
   …                              …
   return &var;                   return &var;
   /* amazingly illegal */        /* (amazingly) legal */
}                             }
```

Static local variables have the same scope as other local variables, but their values are retained across calls to the procedures they are declared in. They are stored in the data section of the address space.

Reconciling Program Scope (1)

tentative definition

(complete) definition

```
int X;

void func1( ...) {
   ...
}
```
file1.c

```
int X=1;

void func2( ...) {
   ...
}
```
file2.c

Where does X go?
What's its initial value?

- tentative definitions overridden by compatible (complete) definitions
- if not overridden, then initial value is zero

X goes in the data section and has an initial value of 1. If file2.c did not exist, then X would go in the bss section and have an initial value of 0. Note that the textbook calls tentative definitions "weak definitions" and complete definitions "strong definitions". This is non-standard terminology and conflicts with the standard use of the term "weak definition," which we discuss shortly.

Reconciling Program Scope (2)

```
int X=2;

void func1( ...) {
   ...
}
```
file1.c

```
int X=1;

void func2( ...) {
   ...
}
```
file2.c

What happens here?

In this case we have conflicting definitions of X — this will be flagged (by the ld program) as an error.

Reconciling Program Scope (3)

```
int X=1;

void func1( ...) {
    ...
}
```
file1.c

```
int X=1;

void func2( ...) {
    ...
}
```
file2.c

Is this ok?

No; it is flagged as an error: only one file may supply an initial value.

Reconciling Program Scope (4)

```
extern int X;

void func1( ...) {
    ...
}
```
file1.c

```
int X=1;

void func2( ...) {
    ...
}
```
file2.c

What's the purpose of "extern"?

The "extern" means that this file will be using X, but it depends on some other file to provide a definition for it, either initialized of uninitialized. If no other file provides a definition, then ld flags an error.

If the "extern" were not there, i.e., if X were declared simply as an "int" in file1.c, then it wouldn't matter if no other file provided a definition for X — X would be allocated in bss with an implicit initial value of 0.

Note: this description of extern is how it is implemented by gcc. The official C99 standard doesn't require this behavior, but merely permits it. It also permits "extern" to be essentially superfluous: its presence may mean the same thing as its absence.

The C11 standard more-or-less agrees with the C99 standard. Moreover, it explicitly allows a declaration of the form "extern int X=1;" (i.e., initialization), which is not allowed by gcc.

For most practical purposes, whatever gcc says is the law ...

Default Values (1)

```
float seed = 1.0;

int PrimaryFunc(int arg) {
  void SecondaryFunc(float arg);
  ...
}

void SecondaryFunc(float arg) {
  ...
}
```

Default Values (2)

```
float seed = 2.0; /* want a different seed */

int main() {
  ...
  PrimaryFunc(floatingValue);
  ...
}

void SecondaryFunc(float arg) {
  /* would like to override default version */
  ...
}
```

The code in this slide will use the code in the previous slide, however, we would like to override the previous slide's definitions of *seed* and *SecondaryFunc*. The linker would not allow this and would flag "duplicate-definition" errors.

Default Values (3)

```
__attribute__((weak)) float seed = 1.0;

int PrimaryFunc(int arg) {
  void SecondaryFunc(float arg);
  ...
}

void __attribute__((weak)) SecondaryFunc(float arg) {
  ...
}
```

By defining *seed* and *SecondaryFunc* to be *weak* symbols, we can indicate that they may be overridden. If there is no other definition for a weak symbol, the "weak" definition will be used. Otherwise the other definition will be used.

Does Location Matter?

```
int main(int argc, char *[]) {
    return(argc);
}
```

```
main:
  pushl %ebp    ; push frame pointer
  movl %esp, %ebp     ; set frame pointer to point to new frame
  movl 8(%ebp), %eax  ; put argc into return register (eax)
  movl %ebp, %esp     ; restore stack pointer
  popl %ebp    ; pop stack into frame pointer
  ret          ; return: pops end of stack into eip
```

This rather trivial IA32 program references memory via only esp and eip (ebp is set from esp). Its code contains no explicit references to memory, i.e., it contains no explicit addresses.

Location Matters …

```
int X=6;
int *aX = &X;

int main() {
    void subr(int);
    int y=*aX;
    subr(y);
    return(0);
}

void subr(int i) {
    printf("i = %d\n", i);
}
```

We don't need to look at the assembler code to see what's different about this program: the machine code produced for it can't simply be copied to an arbitrary location in our computer's memory and executed. The location identified by the name *aX* should contain the address of the location containing *X*. But since the address of *X* will not be known until the program is copied into memory, neither the compiler nor the assembler can initialize *aX* correctly. Similarly, the addresses of *subr* and *printf* are not known until the program is copied into memory — again, neither the compiler nor the assembler would know what addresses to use.

Coping

- **Relocation**
  - modify internal references according to where module is loaded in memory
  - modules needing relocation are said to be *relocatable*
    - » which means they *require* relocation
  - the compiler/assembler provides instructions to the linker on how to do this

A Revision

```
extern int X;
int *aX = &X;
int Y = 1;

int main() {
    void subr(int);
    int y = *aX+Y;
    subr(y);
    return(0);
}
```
main.c

```
#include <stdio.h>
int X;

void subr(int XX) {
    printf("XX = %d\n", XX);
    printf("X = %d\n", X);
}
```
subr.c

```
gcc -o prog -O1 main.c subr.c
```

Note that what we actually did, in order to obtain what's in the next few slides, was:

gcc –S –O1 main.c subr.c
gcc –c main.s subr.s
gcc –o prog main.o subr.o

Note that a symbol's value is the location of what it refers to. The compiler/assembler knows what the values (i.e., locations) of aX and Y are relative to the beginning of this module's data section (next slide), but has no idea what subr's value is. It is the linker's job to provide final values for these symbols, which will be the addresses of the corresponding C constructs when the program is loaded into memory. The linker will adjust these values to obtain the locations of what they refer to relative to the value of register rip when the referencing instructions are executed.

One might ask why these locations are referred to using offsets from the instruction pointer (also known as the program counter), rather than simply using their addresses. The reason is to save space: the addresses would be 64 bits long, but the offsets are only 32 bits long.

The ".file" directive supplies information to be placed in the object file and the executable of use to debuggers — it tells them what the source-code file is.

The ".globl" directive indicates that the symbol, defined here, will be used by other modules, and thus should be made known to the linker.

The ".type" directive indicates how the symbol is used. Two possibilities are function and object (meaning a data object).

The " .size" directive indicates the size that should be associated with the given symbol.

The directives starting with ".cfi_" are there for the sake of the debugger. They generate auxiliary information stored in the object file (but not executed) that describes the relation between the stack pointer (%rsp) and the beginning of the stack frame. Thus they compensate for the lack of a standard frame-pointer register (%esp for IA32). In particular, they emit data going into a table that is used by a debugger (such as gdb) to determine, based on the value of the instruction pointer (%rip) and the stack pointer, where the beginning of the current stack frame is.

main.s (2)

```
            .globl  Y
0:          .data
0:          .align 4
0:          .type   Y, @object
0:          .size   Y, 4
0: Y:
0:          .long   1
4:          .globl  aX
8:          .align 8
8:          .type   aX, @object
8:          .size   aX, 8
8: aX:
8:          .quad     X
8:          .ident  "GCC: (Debian 4.7.2-5) 4.7.2"
0:          .section        .note.GNU-stack,"",@progbits
```

Y should be made known to others

aX should be made known to others

must be replaced with address of X

The symbol *X*'s value is, at this point, unknown.

The ".data" directive indicates that what follows goes in the data section.

The ".long" directive indicates that storage should be allocated for a long word.

The ".quad" directive indicates that storage should be allocated for a quad word.

The ".align" directive indicates that the storage associated with the symbol should be aligned, in the cases here, on 4-byte and 8-byte boundaries (i.e., the least-significant two bits and three bits of their addresses should be zeroes).

The ".ident" directive indicates the software used to produce the file and its version.

The ".section" directive used here is supplied by gcc by default in all cases and indicates that the program should have a non-executable stack.

subr.s (1)

```
            .file   "subr.c"
0:          .section        .rodata.str1.1,"aMS",@progbits,1
0: .LC0:
0:          .string "XX = %d\n"
9: .LC1:
9:          .string "X = %d\n"
```

The ".section" directive here indicates that what follows should be placed in read-only storage (and will be included in the text section). Furthermore, what follows are strings with a one-byte-per-character encoding that require one-byte (i.e., unrestricted) alignment. This information will ultimately be used by the linker to reduce storage by identifying strings that are suffices of others.

subr.s (2)

```
0:           .text
0:           .globl  subr
0:           .type   subr, @function
0: subr:
0: .LFB11:
0:           .cfi_startproc
0:           subq    $8, %rsp
4:           .cfi_def_cfa_offset 16
4:           movl    %edi, %esi
6:           movl    $.LC0, %edi
11:          movl    $0, %eax
16:          call    printf
21:          movl    X(%rip), %esi
27:          movl    $.LC1, %edi
32:          movl    $0, %eax
37:          call    printf
42:          addq    $8, %rsp
46:          .cfi_def_cfa_offset 8
46:          ret
47:          .cfi_endproc
47:.LFE11:
47:          .size   subr, .-subr
```

subr should be made known to others

must be replaced with .LC0's address

must be replaced with .LC1's address

must be replaced with printf's address, expressed as an offset from the next instruction

Note that the compiler has generated *movl* instructions (copying 32 bits) for copying the addresses of .LC0 and .LC1: it's assuming that both addresses will fit in 32 bits (in other words, that the text section of the program will be less than 2^{32} bytes long — probably a reasonable assumption.

subr.s (3)

```
0:          .comm   X,4,4
0:          .ident  "GCC: (Debian 4.7.2-5) 4.7.2"
0:          .section        .note.GNU-stack,"",@progbits
```

The ".comm" directive indicates here that four bytes of four-byte aligned storage are required for X in BSS. "comm" stands for "common", which is what the Fortran language uses to mean the same thing as BSS. Since Fortran predates pretty much everything, its terminology wins (at least here).

ELF

- **Executable and linking format**
  - **used on most Unix systems**
    - » **pretty much all but OS X**
  - **defines format for:**
    - » **.o (object) files**
    - » **.so (shared object) files**
    - » **executable files**

Complete documentation for ELF (much more than you'd ever want to know) can be found at http://refspecs.linuxbase.org/elf/elf.pdf.

Doing Relocation

- **Linker is provided instructions for updating object files**
  - **lots of ways addresses can appear in machine code**
  - **three in common use on x86-64**
    - » **32-bit absolute addresses**
      - **used for text references**
    - » **64-bit absolute addresses**
      - **used for data references**
    - » **32-bit PC-relative addresses**
      - **offset from current value of rip**
      - **used for text and data references**

main.o (1)

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              REL (Relocatable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          0 (bytes into file)
  Start of section headers:          296 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           0 (bytes)
  Number of program headers:         0
  Size of section headers:           64 (bytes)
  Number of section headers:         13
  Section header string table index: 10
```

In this and the next few slides we examine the contents of the object files. This information was obtained by using the program "readelf".

main.o (2)

32-bit, PC-relative address

```
Relocation section '.rela.text' at offset 0x5c0 contains 3 entries:
  Offset          Info           Type            Sym. Value    Sym. Name + Addend
000000000007  000900000002 R_X86_64_PC32    0000000000000008 aX - 4
00000000000f  000a00000002 R_X86_64_PC32    0000000000000000 Y - 4
000000000014  000b00000002 R_X86_64_PC32    0000000000000000 subr - 4

Relocation section '.rela.data' at offset 0x608 contains 1 entries:
  Offset          Info           Type            Sym. Value    Sym. Name + Addend
000000000008  000c00000001 R_X86_64_64      0000000000000000 X + 0
```

64-bit, absolute address

```
  0:   48 03 ec 00            sub    $0x0,%rsp
  4:   48 8b 05 00 00 00 00   mov    0x0(%rip),%rax      # b <main+0xb>
  b:   8b 38                  mov    (%rax),%edi
  d:   03 3d 00 00 00 00      add    0x0(%rip),%edi      # 13 <main+0x13>
 13:   e8 00 00 00 00         callq  18 <main+0x18>
 18:   b8 00 00 00 00         mov    $0x0,%eax
 1d:   48 83 c4 08            add    $0x8,%rsp
 21:   c3                     retq
```

main.o (3)

```
Relocation section '.rela.text' at offset 0x5c0 contains 3 entries:
   Offset          Info           Type           Sym. Value     Sym. Name + Addend
000000000007   000900000002 R_X86_64_PC32    0000000000000008 aX - 4
00000000000f   000a00000002 R_X86_64_PC32    0000000000000000 Y - 4
000000000014   000b00000002 R_X86_64_PC32    0000000000000000 subr - 4

Relocation section '.rela.data' at offset 0x608 contains 1 entries:
   Offset          Info           Type           Sym. Value     Sym. Name + Addend
000000000008   000c00000001 R_X86_64_64      0000000000000000 X + 0

   0:   40 03 ec 08           sub    $0x0,%rsp
   4:   48 8b 05 00 00 00 00  mov    0x0(%rip),%rax        # b <main+0xb>
   b:   8b 38                 mov    (%rax),%edi
   d:   03 3d 00 00 00 00     add    0x0(%rip),%edi        # 13 <main+0x13>
  13:   e8 00 00 00 00        callq  18 <main+0x18>
  18:   b8 00 00 00 00        mov    $0x0,%eax
  1d:   48 83 c4 08           add    $0x8,%rsp
  21:   c3                    retq
```

The first three relocation instructions are for the text associated with main. The first relocation instruction specifies that offset 0x07 of the text region should be updated by adding to it the PC-relative version of the address ultimately associated with symbol aX. This will be, of course, where aX is located in the data region. The field in the "Info" column encodes what's given more clearly in the next three columns. The "0009" identifies a field in the symbol table (not shown) that says the symbol's name is aX and that its value may be found at offset 0x08 (the "Sym. Value" column) in this module's contribution to the data section. The "0002" in the "Info" column says that the type of reference to aX is 32-bit PC-relative (the "Type" column).

To handle PC-relative addressing, the linker blindly assumes that the PC's value (the contents of register rip) is the address of the field within the instruction that's being modified (offset 7 in this example). Thus, for example, if the text section for main were loaded into memory at address 0x1000, the linker would assume that the value contained in register rip would be 0x1007 when the source operand of the first mov instruction is being located. If symbol aX is at, say, location 0x10008, the linker would modify the last four bytes of the mov instruction by replacing its contents with 0xf001 (= 0x10008 – 0x1001). However, by the time rip is used to access the source operand, it will already have been incremented to point to the next instruction (the second mov). If a PC-relative address of 0xf001 were actually used, it would point to four bytes beyond the location of aX. So, to correct for this, rather than use the value of symbol aX directly, the linker is instructed to use four less than this value (hence the "addend" of -4).

main.o (4)

```
Relocation section '.rela.text' at offset 0x5c0 contains 3 entries:
  Offset          Info          Type           Sym. Value    Sym. Name + Addend
000000000007   000900000002 R_X86_64_PC32      0000000000000008 aX - 4
00000000000f   000a00000002 R_X86_64_PC32      0000000000000000 Y - 4
000000000014   000b00000002 R_X86_64_PC32      0000000000000000 subr - 4

Relocation section '.rela.data' at offset 0x608 contains 1 entries:
  Offset          Info          Type           Sym. Value    Sym. Name + Addend
000000000008   000c00000001 R_X86_64_64        0000000000000000 X + 0


  0:   48 03 ec 00            sub    $0x0,%rsp
  4:   48 8b 05 00 00 00 00   mov    0x0(%rip),%rax        # b <main+0xb>
  b:   8b 38                  mov    (%rax),%edi
  d:   03 3d 00 00 00 00      add    0x0(%rip),%edi        # 13 <main+0x13>
 13:   e8 00 00 00 00         callq  18 <main+0x18>
 18:   b8 00 00 00 00         mov    $0x0,%eax
 1d:   48 83 c4 08            add    $0x8,%rsp
 21:   c3                     retq
```

The second relocation instruction specifies that offset 0x0f of the text region should be updated by adding to it the PC-relative version of the address ultimately associated with symbol Y. This will be, of course, where Y is located in the data region.

main.o (4)

```
Relocation section '.rela.text' at offset 0x5c0 contains 3 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000007  000900000002 R_X86_64_PC32    0000000000000008 aX - 4
00000000000f  000a00000002 R_X86_64_PC32    0000000000000000 Y - 4
000000000014  000b00000002 R_X86_64_PC32    0000000000000000 subr - 4

Relocation section '.rela.data' at offset 0x608 contains 1 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000008  000c00000001 R_X86_64_64      0000000000000000 X + 0

   0:   48 03 ec 00           sub    $0x0,%rsp
   4:   48 8b 05 00 00 00 00  mov    0x0(%rip),%rax        # b <main+0xb>
   b:   8b 38                 mov    (%rax),%edi
   d:   03 3d 00 00 00 00     add    0x0(%rip),%edi        # 13 <main+0x13>
  13:   e8 00 00 00 00        callq  18 <main+0x18>
  18:   b8 00 00 00 00        mov    $0x0,%eax
  1d:   48 83 c4 08           add    $0x8,%rsp
  21:   c3                    retq
```

The third relocation instruction specifies that offset 0x14 of the text region should be updated by adding to it the PC-relative version of the address ultimately associated with symbol subr. This will be, of course, where subr is located in the text region.

main.o (5)

```
Relocation section '.rela.text' at offset 0x5c0 contains 3 entries:
  Offset          Info           Type            Sym. Value    Sym. Name + Addend
000000000007   000900000002 R_X86_64_PC32       0000000000000008 aX - 4
00000000000f   000a00000002 R_X86_64_PC32       0000000000000000 Y - 4
000000000014   000b00000002 R_X86_64_PC32       0000000000000000 subr - 4

Relocation section '.rela.data' at offset 0x608 contains 1 entries:
  Offset              Info            Type           Sym. Value     Sym. Name + Addend
000000000008   000c00000001 R_X86_64_64         0000000000000000 X + 0

   0:   48 03 ec 08             sub    $0x0,%rsp
   4:   48 8b 05 00 00 00 00    mov    0x0(%rip),%rax        # b <main+0xb>
   b:   8b 38                   mov    (%rax),%edi
   d:   03 3d 00 00 00 00       add    0x0(%rip),%edi        # 13 <main+0x13>
  13:   e8 00 00 00 00          callq  18 <main+0x18>
  18:   b8 00 00 00 00          mov    $0x0,%eax
  1d:   48 83 c4 08             add    $0x8,%rsp
  21:   c3                      retq
```

The final relocation instruction, which is for the data associated with main, specifies that offset 0x08 of this module's contribution to the data region should be updated by adding to it the address of symbol X, once it's determined.

subr.o (1)

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              REL (Relocatable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          0 (bytes into file)
  Start of section headers:          312 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           0 (bytes)
  Number of program headers:         0
  Size of section headers:           64 (bytes)
  Number of section headers:         13
  Section header string table index: 10
```

subr.o (2)

```
Relocation section '.rela.text' at offset 0x5b0 contains 5 entries:
  Offset          Info            Type           Sym. Value    Sym. Name + Addend
000000000007  00050000000a R_X86_64_32        0000000000000000 .rodata.str1.1 + 0
000000000011  000a00000002 R_X86_64_PC32      0000000000000000 printf - 4
000000000017  000b00000002 R_X86_64_PC32      0000000000000004 X - 4
00000000001c  00050000000a R_X86_64_32        0000000000000000 .rodata.str1.1 + 9
000000000026  000a00000002 R_X86_64_PC32      0000000000000000 printf - 4
```

```
   0:   48 83 ec 08           sub    $0x8,%rsp          .rodata.str1.1:
   4:   89 fe                 mov    %edi,%esi          XX = %d\n\0X = %d\n\0
   6:   bf 00 00 00 00        mov    $0x0,%edi
   b:   b8 00 00 00 00        mov    $0x0,%eax
  10:   e8 00 00 00 00        callq  15 <subr+0x15>
  15:   8b 35 00 00 00 00     mov    0x0(%rip),%esi      # 1b <subr+0x1b>
  1b:   bf 00 00 00 00        mov    $0x0,%edi
  20:   b8 00 00 00 00        mov    $0x0,%eax
  25:   e8 00 00 00 00        callq  2a <subr+0x2a>
  2a:   48 83 c4 08           add    $0x8,%rsp
  2e:   c3                    retq
```

The relocation section for subr includes entries for relocating the references to the strings passed to the calls to printf. For both references, the symbol name is ".rodata.str1.1", which refers to the section containing both strings: the first is at offset 0, the second at offset 9. Hence the addend value is used to indicate which string is being referenced.

printf.o

```
Relocation section '.rela.text' at offset 0x5c0 contains 3 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
0000000002d3  000b00000002 R_X86_64_PC32     0000000000000000 write - 4

Relocation section '.rela.data' at offset 0x608 contains 1 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
0000000000d3  000c00000001 R_X86_64_64       0000000000000000 StandardFiles + 0
```

To simplify our discussion a bit, the version of printf shown here is not what is really provided the C library, but is much simpler. Assume "StandardFiles" is an array of per-file information required by printf (and other I/O routines). Printf calls write, a lower-level I/O routine that we discuss in a later lecture.

prog

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x400400
  Start of program headers:          64 (bytes into file)
  Start of section headers:          2704 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         8
  Size of section headers:           64 (bytes)
  Number of section headers:         31
  Section header string table index: 28
```

This is the ELF header from the final executable created for our fully linked program.

Final Result

Symbol	Value	Size	
_start	0x400400	0x60	
main	0x400460	0x3f	
subr	0x4004a0	0x30	
printf	0x4004d0	0x12000	text
write	0x4124d0	0x30	
.rodata	0x412500	0x9	
aX	0x413000	0x8	
Y	0x413008	0x8	data
StandardFiles	0x413010	0x1000	
X	0x414010	0x8	bss

The slide shows the final layout of the address space. (Though keep in mind that, as already mentioned, what's there for printf is simplified.) Note that a special entry "_start" has been added. This is what is actually called first. It then calls main. When main returns, it returns to _start, which then causes the process to terminate (by calling the operating system's "exit" routine, which we discuss in a later lecture).

If you are exceptionally sharp-eyed, you might notice that .rodata refers to an area that's only 9 bytes long, but that the sum of the lengths of the two format strings passed to the two calls to printf in subr was 17 bytes. The linker actually determines that the second string is a suffix of the first, and thus it's only necessary to store the first (and thus the reference to the second string is a reference to the second character of the first).

One might ask why text starts at 0x400400 (= 4,195,328 in decimal) rather than at a much smaller value (such as 0). The answer is that there's other "stuff" at lower addresses, much of which we'll discuss later. However, it's important that nothing be at location zero (in fact, nothing should be in the first "page" of memory, which is either the first 4k bytes or the first 2M bytes on the x86-64, depending on how configured) — this is so that page can be marked "inaccessible" and thus all attempts to use a zero (null) pointer will fail.