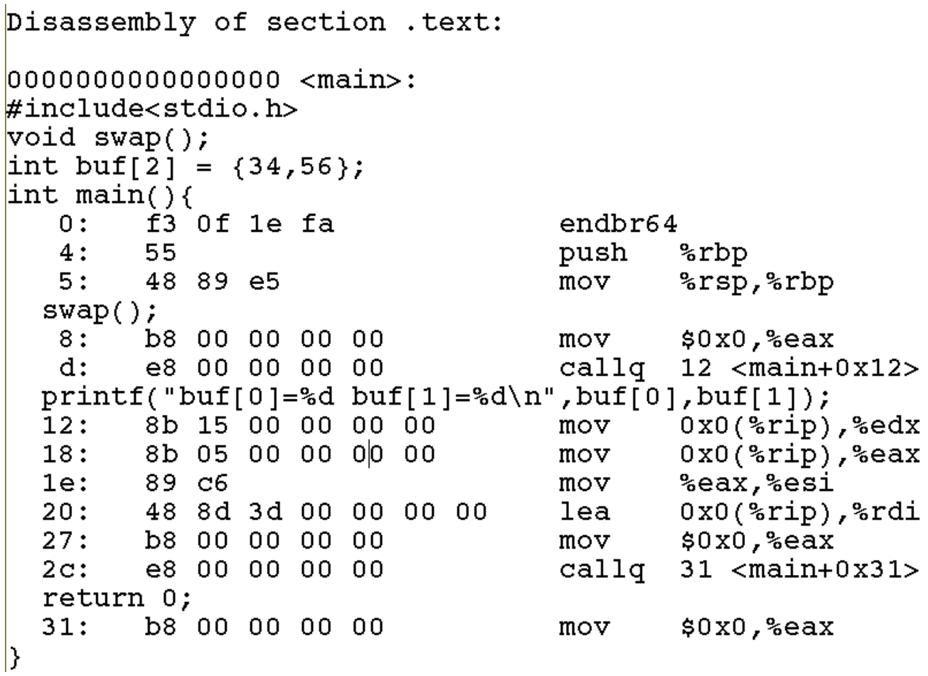
Tutorial 1: Linking and Loading

Model Solutions

Spring 2023

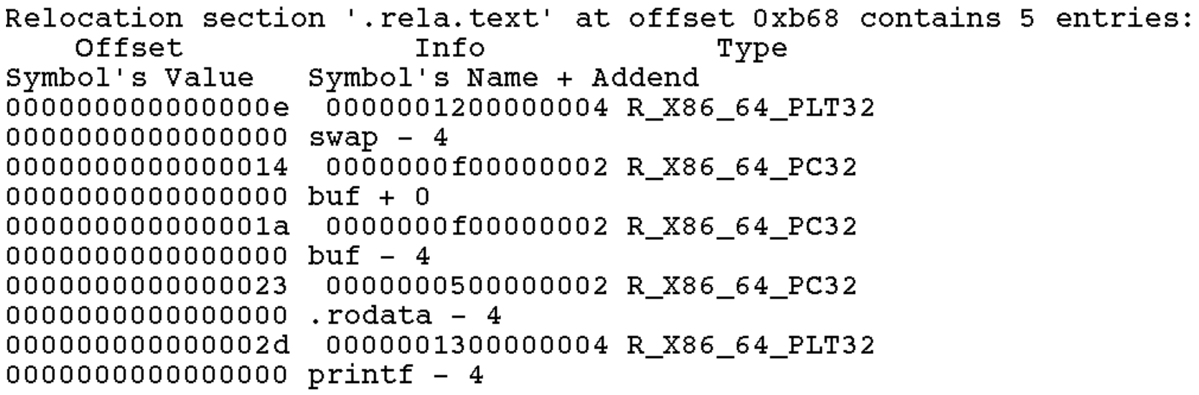
Question 1

Use generate script to generate elf files for the given programs.



```
Disassembly of section .text:

0000000000000000 <main>:
#include<stdio.h>
void swap();
int buf[2] = {34,56};
int main(){
   0:   f3 0f 1e fa             endbr64
   4:   55                      push   %rbp
   5:   48 89 e5                mov    %rsp,%rbp
  swap();
   8:   b8 00 00 00 00          mov    $0x0,%eax
   d:   e8 00 00 00 00          callq  12 <main+0x12>
  printf("buf[0]=%d buf[1]=%d\n",buf[0],buf[1]);
  12:   8b 15 00 00 00 00       mov    0x0(%rip),%edx
  18:   8b 05 00 00 00 00       mov    0x0(%rip),%eax
  1e:   89 c6                   mov    %eax,%esi
  20:   48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi
  27:   b8 00 00 00 00          mov    $0x0,%eax
  2c:   e8 00 00 00 00          callq  31 <main+0x31>
  return 0;
  31:   b8 00 00 00 00          mov    $0x0,%eax
}
```



```
Relocation section '.rela.text' at offset 0xb68 contains 5 entries:
    Offset             Info             Type
Symbol's Value   Symbol's Name + Addend
000000000000000e  0000001200000004 R_X86_64_PLT32
0000000000000000 swap - 4
0000000000000014  0000000f00000002 R_X86_64_PC32
0000000000000000 buf + 0
000000000000001a  0000000f00000002 R_X86_64_PC32
0000000000000000 buf - 4
0000000000000023  0000000500000002 R_X86_64_PC32
0000000000000000 .rodata - 4
000000000000002d  0000001300000004 R_X86_64_PLT32
0000000000000000 printf - 4
```

- A portion of the file main.elf file is shown in the bottom figure. It shows the locations needing relocation in the text segment. Specifically it shows that locations `e`, `14`, `1a`, `23` and `2d` need relocation. No location in the data segment needs relocation.

- It also says that the relocation type is `R_X86_64_PLT32/PC32`. These are PC-relative relocations, and the value to be relocated is a 32 bit quantity in each case.
- Turning to the `main.obj` file on the top: The value to be relocated at `e` stands for the procedure `swap`.

  If *loc* is a location before relocation, let $final(loc)$ denote the final address of *loc* after relocation. Let *offset* be the value that is filled in the 4 bytes starting from `e`. Now the fact that relates these values is that when the call at `d` is executing, the PC is pointing to the next instruction $final(12)$, and the address of `swap` is obtained by adding *offset* to PC.

  $\text{PC} + \mathit{offset} = loc(\texttt{swap})$
  $\mathit{offset} = loc(\texttt{swap}) - \text{PC}$
  $\mathit{offset} = loc(\texttt{swap}) - \text{(loc(e) + 4))}$
  $\mathit{offset} = loc(\texttt{swap}) - loc(\texttt{e})$ - 4

- After the linker has determined the values of $loc(\texttt{swap})$ and $loc(\texttt{e})$, it uses the above formula to obtain the value of *offset*. The value -4 is what is called `Addend` in `main.elf`.

`swap.c` and `others.c` can be analyzed in a similar manner.
There is one more point that needs attention: In `swap.c` there is a location in the data area that needs relocation. This can be seen as an entry in the data relocation area in swap.elf

```
Relocation section '.rela.data.rel' at offset 0x710 contains 1 entry:
    Offset             Info             Type       Symbol's Value   Symbol's Name + Addend
0000000000000000  0000001100000001 R_X86_64_64     0000000000000000 buf + 0
```

This is an absolute relocation. The final address of the target (and not an *offset*) is put directly at the place of relocation. And since this is an address, it is a 64-bit quantity.

Question 2

Annotate fragments of target code with the source statements that they correspond to. Annotate each local variable and parameter with its (relative address).

test1.c

```c
int main()
{
    int a=1, b=1;
    while(a<=10)
    {
        b=b*a;
        a++;
    }
    return b;
}
```

```
main:
    pushq %rbp
    movq %rsp, %rbp
    movl $1, -8(%rbp) -- b=1
    movl $1, -4(%rbp) -- a=1
    jmp .L2 -- while condition
.L3:
    movl -4(%rbp), %eax -- Copy a
    imull -8(%rbp), %eax -- a*b
    movl %eax, -4(%rbp) -- b=b*a
    addl $1, -8(%rbp) -- a++
.L2:
    cmpl $10, -8(%rbp) -- a<=10
    jle .L3 -- to inside while loop
    movl -4(%rbp), %eax -- value of b
    popq %rbp -- return b
    ret
```

Only two local variables, *a* is in $-4(\%rbp)$, or offset -4, *b* is in offset -8.

test2.c

```c
struct data{
    int sum;
    int b[5];
};

int main()
{
    struct data rec1;
    rec1.sum=0;
    rec1.b[0]=2;
    rec1.sum=rec1.sum+
       rec1.b[0];
    return rec1.sum;
}
```

```
main:
    pushq %rbp
    movq %rsp, %rbp
    movl $0, -32(%rbp) -- rec1.sum=0
    movl $2, -28(%rbp) -- rec1.b[0]=2
    movl -32(%rbp), %edx -- rec1.sum
    movl -28(%rbp), %eax -- rec1.b[0]
    addl %edx, %eax -- sum + b[0]
    movl %eax, -32(%rbp) -- rec1.sum
    movl -32(%rbp), %eax -- value of rec1.sum
    popq %rbp -- return rec1.sum
    ret
```

The local variable `rec1` contains sum, which is stored in offset −32, the array `b` spans from offset −28 to −12.

Question 3

a) The `main` in module1. The `main` in module2 is an uninitialized global and therefore a weak symbol.

b) gcc will give an error because both `main`s are strong symbols.

c) No clash of symbols. `main` in module2 is not visible outside of module2.

Question 4

User stack: `p`
Heap: `*p`
Read only: the functions `main` and `f`, the format string `"%d\n"`.
Read write: `x`, `y`, `a`, `k`

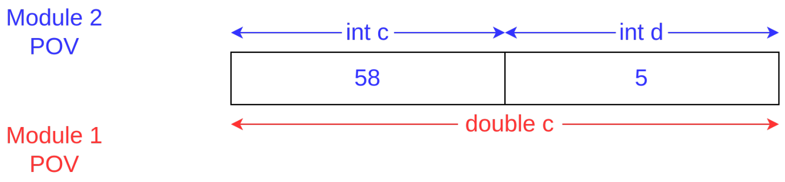
Question 5

The answer is:
$\{14, 3\}$
$\{3, 14\}$
Note that the `rec`s in both modules default to the `rec` in module1. Now for module2, `x` is the second field and `y` is the first field of whatever `rec` resolves to. Thus {14,3}. The printing of {3,14} is obvious.

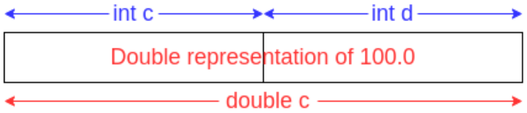
Question 6

The line `int z = a[5]` is a compile time error. `a[5]` is not a compile time constant. However, `int *x = &(a[3])` is ok since `&(a[3])` is a compile time constant.

Now just before the program starts executing, here is a (partial) view of the global memory from the point of view of the two modules:



After the execution of `c = 100.0`, this is what happens:



And since the call to `fn()` returns the `d` part of this memory, it is highly improbable that it is equal to 5.