A. What is the hex address of the relocated reference to `sum` in line 5?

B. What is the hex value of the relocated reference to `sum` in line 5?

Practice Problem 7.5 (solution page 754)

Consider the call to function `swap` in object file `m.o` (Figure 7.5).

```
   9:   e8 00 00 00 00          callq  e <main+0xe>      swap()
```

with the following relocation entry:

```
r.offset = 0xa
r.symbol = swap
r.type   = R_X86_64_PC32
r.addend = -4
```

Now suppose that the linker relocates `.text` in `m.o` to address `0x4004d0` and `swap` to address `0x4004e8`. Then what is the value of the relocated reference to `swap` in the `callq` instruction?

7.8 Executable Object Files

We have seen how the linker merges multiple object files into a single executable object file. Our example C program, which began life as a collection of ASCII text files, has been transformed into a single binary file that contains all of the information needed to load the program into memory and run it. Figure 7.13 summarizes the kinds of information in a typical ELF executable file.

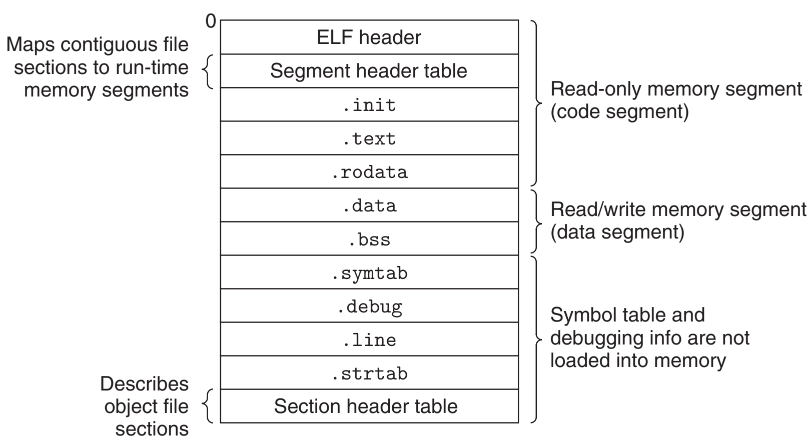

Figure 7.13 **Typical ELF executable object file.**

code/link/prog-exe.d

```
    Read-only code segment
1   LOAD off    0x0000000000000000 vaddr 0x0000000000400000 paddr 0x0000000000400000 align 2**21
2        filesz 0x000000000000069c memsz 0x000000000000069c flags r-x

    Read/write data segment
3   LOAD off    0x0000000000000df8 vaddr 0x0000000000600df8 paddr 0x0000000000600df8 align 2**21
4        filesz 0x0000000000000228 memsz 0x0000000000000230 flags rw-
```

code/link/prog-exe.d

Figure 7.14 **Program header table for the example executable** `prog`. `off`: offset in object file; `vaddr/paddr`: memory address; `align`: alignment requirement; `filesz`: segment size in object file; `memsz`: segment size in memory; `flags`: run-time permissions.

The format of an executable object file is similar to that of a relocatable object file. The ELF header describes the overall format of the file. It also includes the program's *entry point*, which is the address of the first instruction to execute when the program runs. The `.text`, `.rodata`, and `.data` sections are similar to those in a relocatable object file, except that these sections have been relocated to their eventual run-time memory addresses. The `.init` section defines a small function, called `_init`, that will be called by the program's initialization code. Since the executable is *fully linked* (relocated), it needs no `.rel` sections.

ELF executables are designed to be easy to load into memory, with contiguous chunks of the executable file mapped to contiguous memory segments. This mapping is described by the *program header table*. Figure 7.14 shows part of the program header table for our example executable `prog`, as displayed by OBJDUMP.

From the program header table, we see that two memory segments will be initialized with the contents of the executable object file. Lines 1 and 2 tell us that the first segment (the *code segment*) has read/execute permissions, starts at memory address `0x400000`, has a total size in memory of `0x69c` bytes, and is initialized with the first `0x69c` bytes of the executable object file, which includes the ELF header, the program header table, and the `.init`, `.text`, and `.rodata` sections.

Lines 3 and 4 tell us that the second segment (the *data segment*) has read/write permissions, starts at memory address `0x600df8`, has a total memory size of `0x230` bytes, and is initialized with the `0x228` bytes in the `.data` section starting at offset `0xdf8` in the object file. The remaining 8 bytes in the segment correspond to `.bss` data that will be initialized to zero at run time.

For any segment s, the linker must choose a starting address, `vaddr`, such that

$$\texttt{vaddr} \bmod \texttt{align} = \texttt{off} \bmod \texttt{align}$$

where `off` is the offset of the segment's first section in the object file, and `align` is the alignment specified in the program header ($2^{21} = \texttt{0x200000}$). For example, in the data segment in Figure 7.14,

$$\texttt{vaddr} \bmod \texttt{align} = \texttt{0x600df8} \bmod \texttt{0x200000} = \texttt{0xdf8}$$

and

$$\texttt{off} \bmod \texttt{align} = \texttt{0xdf8} \bmod \texttt{0x200000} = \texttt{0xdf8}$$

This alignment requirement is an optimization that enables segments in the object file to be transferred efficiently to memory when the program executes. The reason is somewhat subtle and is due to the way that virtual memory is organized as large contiguous power-of-2 chunks of bytes. You will learn all about virtual memory in Chapter 9.

7.9 Loading Executable Object Files

To run an executable object file `prog`, we can type its name to the Linux shell's command line:

```
linux> ./prog
```

Since `prog` does not correspond to a built-in shell command, the shell assumes that `prog` is an executable object file, which it runs for us by invoking some memory-resident operating system code known as the `loader`. Any Linux program can invoke the loader by calling the `execve` function, which we will describe in detail in Section 8.4.6. The loader copies the code and data in the executable object file from disk into memory and then runs the program by jumping to its first instruction, or *entry point*. This process of copying the program into memory and then running it is known as *loading*.

Every running Linux program has a run-time memory image similar to the one in Figure 7.15. On Linux x86-64 systems, the code segment starts at address `0x400000`, followed by the data segment. The run-time *heap* follows the data segment and grows upward via calls to the `malloc` library. (We will describe `malloc` and the heap in detail in Section 9.9.) This is followed by a region that is reserved for shared modules. The user stack starts below the largest legal user address ($2^{48} - 1$) and grows down, toward smaller memory addresses. The region above the stack, starting at address 2^{48}, is reserved for the code and data in the *kernel*, which is the memory-resident part of the operating system.

For simplicity, we've drawn the heap, data, and code segments as abutting each other, and we've placed the top of the stack at the largest legal user address. In practice, there is a gap between the code and data segments due to the alignment requirement on the `.data` segment (Section 7.8). Also, the linker uses address-space layout randomization (ASLR, Section 3.10.4) when it assigns run-time addresses to the stack, shared library, and heap segments. Even though the locations of these regions change each time the program is run, their relative positions are the same.

When the loader runs, it creates a memory image similar to the one shown in Figure 7.15. Guided by the program header table, it copies chunks of the