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

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

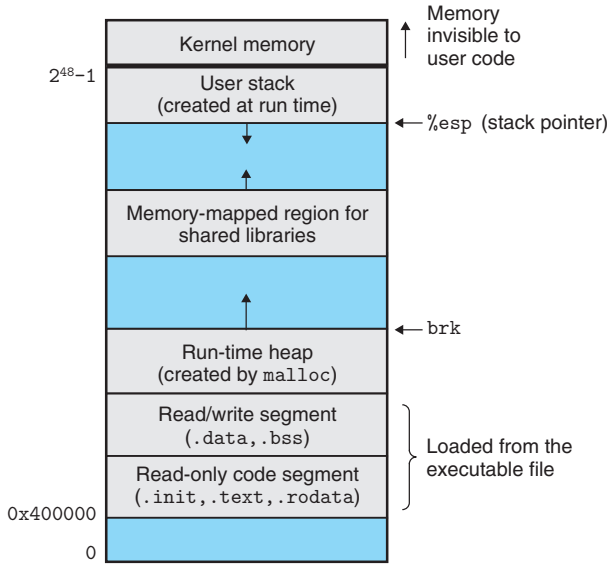

Figure 7.15
Linux x86-64 run-time memory image. Gaps due to segment alignment requirements and address-space layout randomization (ASLR) are not shown. Not to scale.

executable object file into the code and data segments. Next, the loader jumps to the program's entry point, which is always the address of the `_start` function. This function is defined in the system object file `crt1.o` and is the same for all C programs. The `_start` function calls the *system startup function*, `__libc_start_main`, which is defined in `libc.so`. It initializes the execution environment, calls the user-level `main` function, handles its return value, and if necessary returns control to the kernel.

7.10 Dynamic Linking with Shared Libraries

The static libraries that we studied in Section 7.6.2 address many of the issues associated with making large collections of related functions available to application programs. However, static libraries still have some significant disadvantages. Static libraries, like all software, need to be maintained and updated periodically. If application programmers want to use the most recent version of a library, they must somehow become aware that the library has changed and then explicitly relink their programs against the updated library.

Another issue is that almost every C program uses standard I/O functions such as `printf` and `scanf`. At run time, the code for these functions is duplicated in the text segment of each running process. On a typical system that is running hundreds of processes, this can be a significant waste of scarce memory system resources. (An interesting property of memory is that it is *always* a scarce resource, regardless