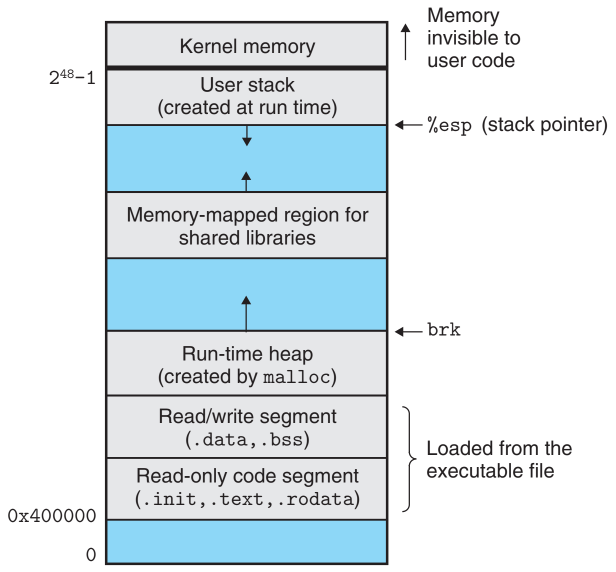

Figure 7.15
Linux x86-64 run-time memory image. Gaps due to segment alignment requirements and address-space layout randomization (ASLR) are not shown. Not to scale.

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

Aside How do loaders really work?

Our description of loading is conceptually correct but intentionally not entirely accurate. To understand how loading really works, you must understand the concepts of *processes*, *virtual memory*, and *memory mapping*, which we haven't discussed yet. As we encounter these concepts later in Chapters 8 and 9, we will revisit loading and gradually reveal the mystery to you.

For the impatient reader, here is a preview of how loading really works: Each program in a Linux system runs in the context of a process with its own virtual address space. When the shell runs a program, the parent shell process forks a child process that is a duplicate of the parent. The child process invokes the loader via the `execve` system call. The loader deletes the child's existing virtual memory segments and creates a new set of code, data, heap, and stack segments. The new stack and heap segments are initialized to zero. The new code and data segments are initialized to the contents of the executable file by mapping pages in the virtual address space to page-size chunks of the executable file. Finally, the loader jumps to the `_start` address, which eventually calls the application's `main` routine. Aside from some header information, there is no copying of data from disk to memory during loading. The copying is deferred until the CPU references a mapped virtual page, at which point the operating system automatically transfers the page from disk to memory using its paging mechanism.

of how much there is in a system. Disk space and kitchen trash cans share this same property.)

Shared libraries are modern innovations that address the disadvantages of static libraries. A shared library is an object module that, at either run time or load time, can be loaded at an arbitrary memory address and linked with a program in memory. This process is known as *dynamic linking* and is performed by a program called a *dynamic linker*. Shared libraries are also referred to as *shared objects*, and on Linux systems they are indicated by the `.so` suffix. Microsoft operating systems make heavy use of shared libraries, which they refer to as DLLs (dynamic link libraries).

Shared libraries are "shared" in two different ways. First, in any given file system, there is exactly one `.so` file for a particular library. The code and data in this `.so` file are shared by all of the executable object files that reference the library, as opposed to the contents of static libraries, which are copied and embedded in the executables that reference them. Second, a single copy of the `.text` section of a shared library in memory can be shared by different running processes. We will explore this in more detail when we study virtual memory in Chapter 9.

Figure 7.16 summarizes the dynamic linking process for the example program in Figure 7.7. To build a shared library `libvector.so` of our example vector routines in Figure 7.6, we invoke the compiler driver with some special directives to the compiler and linker:

```
linux> gcc -shared -fpic -o libvector.so addvec.c multvec.c
```

The `-fpic` flag directs the compiler to generate *position-independent code* (more on this in the next section). The `-shared` flag directs the linker to create a shared

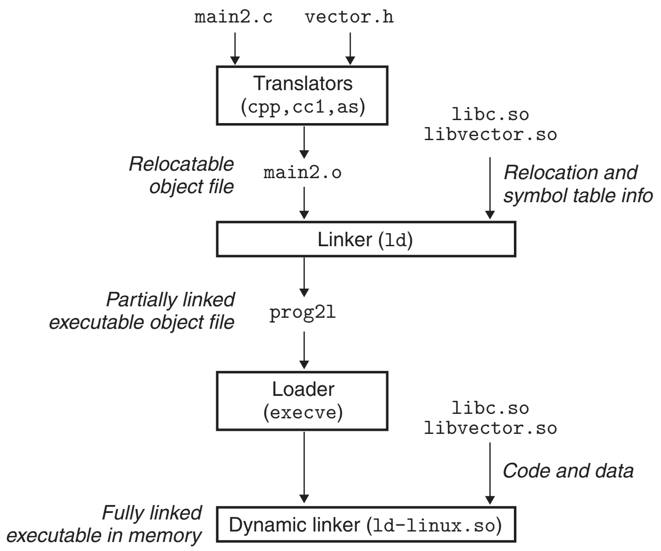

Figure 7.16
Dynamic linking with shared libraries.

object file. Once we have created the library, we would then link it into our example program in Figure 7.7:

```
linux> gcc -o prog2l main2.c ./libvector.so
```

This creates an executable object file `prog2l` in a form that can be linked with `libvector.so` at run time. The basic idea is to do some of the linking statically when the executable file is created, and then complete the linking process dynamically when the program is loaded. It is important to realize that none of the code or data sections from `libvector.so` are actually copied into the executable `prog2l` at this point. Instead, the linker copies some relocation and symbol table information that will allow references to code and data in `libvector.so` to be resolved at load time.

When the loader loads and runs the executable `prog2l`, it loads the partially linked executable `prog2l`, using the techniques discussed in Section 7.9. Next, it notices that `prog2l` contains a `.interp` section, which contains the path name of the dynamic linker, which is itself a shared object (e.g., `ld-linux.so` on Linux systems). Instead of passing control to the application, as it would normally do, the loader loads and runs the dynamic linker. The dynamic linker then finishes the linking task by performing the following relocations:

- Relocating the text and data of `libc.so` into some memory segment
- Relocating the text and data of `libvector.so` into another memory segment
- Relocating any references in `prog2l` to symbols defined by `libc.so` and `libvector.so`

Finally, the dynamic linker passes control to the application. From this point on, the locations of the shared libraries are fixed and do not change during execution of the program.

7.11 Loading and Linking Shared Libraries from Applications

Up to this point, we have discussed the scenario in which the dynamic linker loads and links shared libraries when an application is loaded, just before it executes. However, it is also possible for an application to request the dynamic linker to load and link arbitrary shared libraries while the application is running, without having to link in the applications against those libraries at compile time.

Dynamic linking is a powerful and useful technique. Here are some examples in the real world:

- *Distributing software.* Developers of Microsoft Windows applications frequently use shared libraries to distribute software updates. They generate a new copy of a shared library, which users can then download and use as a replacement for the current version. The next time they run their application, it will automatically link and load the new shared library.
- *Building high-performance Web servers.* Many Web servers generate *dynamic content*, such as personalized Web pages, account balances, and banner ads. Early Web servers generated dynamic content by using `fork` and `execve` to create a child process and run a "CGI program" in the context of the child. However, modern high-performance Web servers can generate dynamic content using a more efficient and sophisticated approach based on dynamic linking.

  The idea is to package each function that generates dynamic content in a shared library. When a request arrives from a Web browser, the server dynamically loads and links the appropriate function and then calls it directly, as opposed to using `fork` and `execve` to run the function in the context of a child process. The function remains cached in the server's address space, so subsequent requests can be handled at the cost of a simple function call. This can have a significant impact on the throughput of a busy site. Further, existing functions can be updated and new functions can be added at run time, without stopping the server.

Linux systems provide a simple interface to the dynamic linker that allows application programs to load and link shared libraries at run time.

```
#include <dlfcn.h>

void *dlopen(const char *filename, int flag);
```

Returns: pointer to handle if OK, NULL on error