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

The dlopen function loads and links the shared library filename. The external symbols in filename are resolved using libraries previously opened with the RTLD_GLOBAL flag. If the current executable was compiled with the -rdynamic flag, then its global symbols are also available for symbol resolution. The flag argument must include either RTLD_NOW, which tells the linker to resolve references to external symbols immediately, or the RTLD_LAZY flag, which instructs the linker to defer symbol resolution until code from the library is executed. Either of these values can be ORed with the RTLD_GLOBAL flag.

```
#include <dlfcn.h>

void *dlsym(void *handle, char *symbol);
```

Returns: pointer to symbol if OK, NULL on error

The dlsym function takes a handle to a previously opened shared library and a symbol name and returns the address of the symbol, if it exists, or NULL otherwise.

```
#include <dlfcn.h>

int dlclose (void *handle);
```

Returns: 0 if OK, −1 on error

The dlclose function unloads the shared library if no other shared libraries are still using it.

```
#include <dlfcn.h>

const char *dlerror(void);
```

Returns: error message if previous call to dlopen, dlsym, or dlclose failed;
NULL if previous call was OK

The dlerror function returns a string describing the most recent error that occurred as a result of calling dlopen, dlsym, or dlclose, or NULL if no error occurred.

Figure 7.17 shows how we would use this interface to dynamically link our libvector.so shared library at run time and then invoke its addvec routine. To compile the program, we would invoke GCC in the following way:

```
linux> gcc -rdynamic -o prog2r dll.c -ldl
```

code/link/dll.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

int x[2] = {1, 2};
int y[2] = {3, 4};
int z[2];

int main()
{
    void *handle;
    void (*addvec)(int *, int *, int *, int);
    char *error;

    /* Dynamically load the shared library containing addvec() */
    handle = dlopen("./libvector.so", RTLD_LAZY);
    if (!handle) {
        fprintf(stderr, "%s\n", dlerror());
        exit(1);
    }

    /* Get a pointer to the addvec() function we just loaded */
    addvec = dlsym(handle, "addvec");
    if ((error = dlerror()) != NULL) {
        fprintf(stderr, "%s\n", error);
        exit(1);
    }

    /* Now we can call addvec() just like any other function */
    addvec(x, y, z, 2);
    printf("z = [%d %d]\n", z[0], z[1]);

    /* Unload the shared library */
    if (dlclose(handle) < 0) {
        fprintf(stderr, "%s\n", dlerror());
        exit(1);
    }
    return 0;
}
```

code/link/dll.c

Figure 7.17 **Example program 3.** Dynamically loads and links the shared library `libvector.so` at run time.

Aside Shared libraries and the Java Native Interface

Java defines a standard calling convention called *Java Native Interface (JNI)* that allows "native" C and C++ functions to be called from Java programs. The basic idea of JNI is to compile the native C function, say, `foo`, into a shared library, say, `foo.so`. When a running Java program attempts to invoke function `foo`, the Java interpreter uses the `dlopen` interface (or something like it) to dynamically link and load `foo.so` and then call `foo`.

7.12 Position-Independent Code (PIC)

A key purpose of shared libraries is to allow multiple running processes to share the same library code in memory and thus save precious memory resources. So how can multiple processes share a single copy of a program? One approach would be to assign a priori a dedicated chunk of the address space to each shared library, and then require the loader to always load the shared library at that address. While straightforward, this approach creates some serious problems. It would be an inefficient use of the address space because portions of the space would be allocated even if a process didn't use the library. It would also be difficult to manage. We would have to ensure that none of the chunks overlapped. Each time a library was modified, we would have to make sure that it still fit in its assigned chunk. If not, then we would have to find a new chunk. And if we created a new library, we would have to find room for it. Over time, given the hundreds of libraries and versions of libraries in a system, it would be difficult to keep the address space from fragmenting into lots of small unused but unusable holes. Even worse, the assignment of libraries to memory would be different for each system, thus creating even more management headaches.

To avoid these problems, modern systems compile the code segments of shared modules so that they can be loaded anywhere in memory without having to be modified by the linker. With this approach, a single copy of a shared module's code segment can be shared by an unlimited number of processes. (Of course, each process will still get its own copy of the read/write data segment.)

Code that can be loaded without needing any relocations is known as *position-independent code (PIC)*. Users direct GNU compilation systems to generate PIC code with the `-fpic` option to GCC. Shared libraries must always be compiled with this option.

On x86-64 systems, references to symbols in the same executable object module require no special treatment to be PIC. These references can be compiled using PC-relative addressing and relocated by the static linker when it builds the object file. However, references to external procedures and global variables that are defined by shared modules require some special techniques, which we describe next.

PIC Data References

Compilers generate PIC references to global variables by exploiting the following interesting fact: no matter where we load an object module (including shared