these entries is responsible for invoking a specific function. `PLT[1]` (not shown here) invokes the system startup function (`__libc_start_main`), which initializes the execution environment, calls the `main` function, and handles its return value. Entries starting at `PLT[2]` invoke functions called by the user code. In our example, `PLT[2]` invokes `addvec` and `PLT[3]` (not shown) invokes `printf`.

Global offset table (GOT). As we have seen, the GOT is an array of 8-byte address entries. When used in conjunction with the PLT, `GOT[0]` and `GOT[1]` contain information that the dynamic linker uses when it resolves function addresses. `GOT[2]` is the entry point for the dynamic linker in the `ld-linux.so` module. Each of the remaining entries corresponds to a called function whose address needs to be resolved at run time. Each has a matching PLT entry. For example, `GOT[4]` and `PLT[2]` correspond to `addvec`. Initially, each GOT entry points to the second instruction in the corresponding PLT entry.

Figure 7.19(a) shows how the GOT and PLT work together to lazily resolve the run-time address of function `addvec` the first time it is called:

Step 1. Instead of directly calling `addvec`, the program calls into `PLT[2]`, which is the PLT entry for `addvec`.

Step 2. The first PLT instruction does an indirect jump through `GOT[4]`. Since each GOT entry initially points to the second instruction in its corresponding PLT entry, the indirect jump simply transfers control back to the next instruction in `PLT[2]`.

Step 3. After pushing an ID for `addvec` (`0x1`) onto the stack, `PLT[2]` jumps to `PLT[0]`.

Step 4. `PLT[0]` pushes an argument for the dynamic linker indirectly through `GOT[1]` and then jumps into the dynamic linker indirectly through `GOT[2]`. The dynamic linker uses the two stack entries to determine the run-time location of `addvec`, overwrites `GOT[4]` with this address, and passes control to `addvec`.

Figure 7.19(b) shows the control flow for any subsequent invocations of `addvec`:

Step 1. Control passes to `PLT[2]` as before.

Step 2. However, this time the indirect jump through `GOT[4]` transfers control directly to `addvec`.

7.13 Library Interpositioning

Linux linkers support a powerful technique, called *library interpositioning*, that allows you to intercept calls to shared library functions and execute your own code instead. Using interpositioning, you could trace the number of times a particular library function is called, validate and trace its input and output values, or even replace it with a completely different implementation.

Here's the basic idea: Given some *target function* to be interposed on, you create a *wrapper function* whose prototype is identical to the target function. Using some particular interpositioning mechanism, you then trick the system into calling the wrapper function instead of the target function. The wrapper function typically executes its own logic, then calls the target function and passes its return value back to the caller.

Interpositioning can occur at compile time, link time, or run time as the program is being loaded and executed. To explore these different mechanisms, we will use the example program in Figure 7.20(a) as a running example. It calls the `malloc` and `free` functions from the C standard library (`libc.so`). The call to `malloc` allocates a block of 32 bytes from the heap and returns a pointer to the block. The call to `free` gives the block back to the heap, for use by subsequent calls to `malloc`. Our goal is to use interpositioning to trace the calls to `malloc` and `free` as the program runs.

7.13.1 Compile-Time Interpositioning

Figure 7.20 shows how to use the C preprocessor to interpose at compile time. Each wrapper function in `mymalloc.c` (Figure 7.20(c)) calls the target function, prints a trace, and returns. The local `malloc.h` header file (Figure 7.20(b)) instructs the preprocessor to replace each call to a target function with a call to its wrapper. Here is how to compile and link the program:

```
linux> gcc -DCOMPILETIME -c mymalloc.c
linux> gcc -I. -o intc int.c mymalloc.o
```

The interpositioning happens because of the `-I.` argument, which tells the C preprocessor to look for `malloc.h` in the current directory before looking in the usual system directories. Notice that the wrappers in `mymalloc.c` are compiled with the standard `malloc.h` header file.

Running the program gives the following trace:

```
linux> ./intc
malloc(32)=0x9ee010
free(0x9ee010)
```

7.13.2 Link-Time Interpositioning

The Linux static linker supports link-time interpositioning with the `--wrap f` flag. This flag tells the linker to resolve references to symbol `f` as `__wrap_f` (two underscores for the prefix), and to resolve references to symbol `__real_f` (two underscores for the prefix) as `f`. Figure 7.21 shows the wrappers for our example program.

Here is how to compile the source files into relocatable object files:

```
linux> gcc -DLINKTIME -c mymalloc.c
linux> gcc -c int.c
```

(a) Example program `int.c`

code/link/interpose/int.c

```c
#include <stdio.h>
#include <malloc.h>

int main()
{
    int *p = malloc(32);
    free(p);
    return(0);
}
```

code/link/interpose/int.c

(b) Local `malloc.h` file

code/link/interpose/malloc.h

```c
#define malloc(size) mymalloc(size)
#define free(ptr) myfree(ptr)

void *mymalloc(size_t size);
void myfree(void *ptr);
```

code/link/interpose/malloc.h

(c) Wrapper functions in `mymalloc.c`

code/link/interpose/mymalloc.c

```c
#ifdef COMPILETIME
#include <stdio.h>
#include <malloc.h>

/* malloc wrapper function */
void *mymalloc(size_t size)
{
    void *ptr = malloc(size);
    printf("malloc(%d)=%p\n",
           (int)size, ptr);
    return ptr;
}

/* free wrapper function */
void myfree(void *ptr)
{
    free(ptr);
    printf("free(%p)\n", ptr);
}
#endif
```

code/link/interpose/mymalloc.c

Figure 7.20 Compile-time interpositioning with the C preprocessor.

code/link/interpose/mymalloc.c

```
#ifdef LINKTIME
#include <stdio.h>

void *__real_malloc(size_t size);
void __real_free(void *ptr);

/* malloc wrapper function */
void *__wrap_malloc(size_t size)
{
    void *ptr = __real_malloc(size); /* Call libc malloc */
    printf("malloc(%d) = %p\n", (int)size, ptr);
    return ptr;
}

/* free wrapper function */
void __wrap_free(void *ptr)
{
    __real_free(ptr); /* Call libc free */
    printf("free(%p)\n", ptr);
}
#endif
```

code/link/interpose/mymalloc.c

Figure 7.21 Link-time interpositioning with the `--wrap` flag.

And here is how to link the object files into an executable:

```
linux> gcc -Wl,--wrap,malloc -Wl,--wrap,free -o intl int.o mymalloc.o
```

The `-Wl,option` flag passes `option` to the linker. Each comma in `option` is replaced with a space. So `-Wl,--wrap,malloc` passes `--wrap malloc` to the linker, and similarly for `-Wl,--wrap,free`.

Running the program gives the following trace:

```
linux> ./intl
malloc(32) = 0x18cf010
free(0x18cf010)
```

7.13.3 Run-Time Interpositioning

Compile-time interpositioning requires access to a program's source files. Link-time interpositioning requires access to its relocatable object files. However, there is a mechanism for interpositioning at run time that requires access only to the executable object file. This fascinating mechanism is based on the dynamic linker's `LD_PRELOAD` environment variable.

If the LD_PRELOAD environment variable is set to a list of shared library pathnames (separated by spaces or colons), then when you load and execute a program, the dynamic linker (LD-LINUX.SO) will search the LD_PRELOAD libraries first, before any other shared libraries, when it resolves undefined references. With this mechanism, you can interpose on any function in any shared library, including libc.so, when you load and execute any executable.

Figure 7.22 shows the wrappers for malloc and free. In each wrapper, the call to dlsym returns the pointer to the target libc function. The wrapper then calls the target function, prints a trace, and returns.

Here is how to build the shared library that contains the wrapper functions:

```
linux> gcc -DRUNTIME -shared -fpic -o mymalloc.so mymalloc.c -ldl
```

Here is how to compile the main program:

```
linux> gcc -o intr int.c
```

Here is how to run the program from the bash shell:[3]

```
linux> LD_PRELOAD="./mymalloc.so" ./intr
malloc(32) = 0x1bf7010
free(0x1bf7010)
```

And here is how to run it from the csh or tcsh shells:

```
linux> (setenv LD_PRELOAD "./mymalloc.so"; ./intr; unsetenv LD_PRELOAD)
malloc(32) = 0x2157010
free(0x2157010)
```

Notice that you can use LD_PRELOAD to interpose on the library calls of *any* executable program!

```
linux> LD_PRELOAD="./mymalloc.so" /usr/bin/uptime
malloc(568) = 0x21bb010
free(0x21bb010)
malloc(15) = 0x21bb010
malloc(568) = 0x21bb030
malloc(2255) = 0x21bb270
free(0x21bb030)
malloc(20) = 0x21bb030
malloc(20) = 0x21bb050
malloc(20) = 0x21bb070
malloc(20) = 0x21bb090
malloc(20) = 0x21bb0b0
malloc(384) = 0x21bb0d0
 20:47:36 up 85 days,  6:04,  1 user,  load average: 0.10, 0.04, 0.05
```

3. If you don't know what shell you are running, type printenv SHELL at the command line.

code/link/interpose/mymalloc.c

```
#ifdef RUNTIME
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

/* malloc wrapper function */
void *malloc(size_t size)
{
    void *(*mallocp)(size_t size);
    char *error;

    mallocp = dlsym(RTLD_NEXT, "malloc"); /* Get address of libc malloc */
    if ((error = dlerror()) != NULL) {
        fputs(error, stderr);
        exit(1);
    }
    char *ptr = mallocp(size); /* Call libc malloc */
    printf("malloc(%d) = %p\n", (int)size, ptr);
    return ptr;
}

/* free wrapper function */
void free(void *ptr)
{
    void (*freep)(void *) = NULL;
    char *error;

    if (!ptr)
        return;

    freep = dlsym(RTLD_NEXT, "free"); /* Get address of libc free */
    if ((error = dlerror()) != NULL) {
        fputs(error, stderr);
        exit(1);
    }
    freep(ptr); /* Call libc free */
    printf("free(%p)\n", ptr);
}
#endif
```

code/link/interpose/mymalloc.c

Figure 7.22 Run-time interpositioning with `LD_PRELOAD`.

7.14 Tools for Manipulating Object Files

There are a number of tools available on Linux systems to help you understand and manipulate object files. In particular, the GNU *binutils* package is especially helpful and runs on every Linux platform.

AR. Creates static libraries, and inserts, deletes, lists, and extracts members.

STRINGS. Lists all of the printable strings contained in an object file.

STRIP. Deletes symbol table information from an object file.

NM. Lists the symbols defined in the symbol table of an object file.

SIZE. Lists the names and sizes of the sections in an object file.

READELF. Displays the complete structure of an object file, including all of the information encoded in the ELF header. Subsumes the functionality of SIZE and NM.

OBJDUMP. The mother of all binary tools. Can display all of the information in an object file. Its most useful function is disassembling the binary instructions in the `.text` section.

Linux systems also provide the LDD program for manipulating shared libraries:

LDD: Lists the shared libraries that an executable needs at run time.

7.15 Summary

Linking can be performed at compile time by static linkers and at load time and run time by dynamic linkers. Linkers manipulate binary files called object files, which come in three different forms: relocatable, executable, and shared. Relocatable object files are combined by static linkers into an executable object file that can be loaded into memory and executed. Shared object files (shared libraries) are linked and loaded by dynamic linkers at run time, either implicitly when the calling program is loaded and begins executing, or on demand, when the program calls functions from the `dlopen` library.

The two main tasks of linkers are symbol resolution, where each global symbol in an object file is bound to a unique definition, and relocation, where the ultimate memory address for each symbol is determined and where references to those objects are modified.

Static linkers are invoked by compiler drivers such as GCC. They combine multiple relocatable object files into a single executable object file. Multiple object files can define the same symbol, and the rules that linkers use for silently resolving these multiple definitions can introduce subtle bugs in user programs.

Multiple object files can be concatenated in a single static library. Linkers use libraries to resolve symbol references in other object modules. The left-to-right sequential scan that many linkers use to resolve symbol references is another source of confusing link-time errors.