Intercepting Arbitrary Functions on Windows, UNIX, and Macintosh OS X Platforms

CS-TR-4585, UMIACS-TR-2004-28

Daniel S. Myers and Adam L. Bazinet[*]
Center for Bioinformatics and Computational Biology
Institute for Advanced Computer Studies, University of Maryland, MD 20742 USA

Abstract

It is often desirable to modify the behavior of existing code bases by wrapping or replacing functions. When editing the source code of those functions is a viable option, this can be a straight-forward process. When the source of the functions cannot be edited (e.g., if the functions are provided by the system C library), then alternative techniques are required. Here, we present such techniques for UNIX, Windows, and Macintosh OS X platforms. We have used these techniques to update bioinformatics applications to call the application program interface (API) provided by the Berkeley Open Infrastructure for Network Computing (BOINC), a distributed computing toolkit.

[*]Contact Michael P. Cummings (mike@umiacs.umd.edu) for correspondence.

Introduction

Oftentimes, it can be useful to modify the behavior of an application without making extensive changes to the source code of the application. Specifically, one might want to intercept calls to certain functions so as to execute custom code before and after the execution of each function. For example, it might be necessary to instrument the application for performance analysis or to update it to call some new application program interface (API). In those cases where one can edit the source of the functions in question, source-level editing can be a viable option. In many cases, however, editing the source code for a function is not practical (e.g., the functions in question may be provided by the system C library). In such cases, alternative techniques are required.

In our case, we wanted to run a number of legacy bioinformatics codes under the Berkeley Open Infrastructure for Network Computing (BOINC) [2], which is a toolkit for generalized master/worker Internet-scale public computing in the SETI@home model [1]. Under BOINC, all data files are renamed so as to have globally-unique names, and thus applications doing disk access are required to call a BOINC API function, `boinc_resolve_filename`, to map between the filenames the application expects to use and the unique filenames used by BOINC. In our case, we had a large number of legacy codes which we wanted to port, and we wanted to avoid manually modifying every call to `fopen`, `stat`, and so on to make the necessary API call. We thus found ourselves in need of function-trapping techniques for the UNIX, Macintosh OS X, and Windows platforms on which BOINC can run.

Here, we present the techniques which we used. Please note that these techniques are not novel, and we do not claim that they represent the best solutions for all cases. They have worked well for us, and we present them in the hopes that they may be of use to other groups facing similar situations.

UNIX platforms with GNU LD

There are at least two ways to intercept function calls on UNIX platforms that use the GNU linker: one may use the `LD_PRELOAD` environment variable at run-time to load a shared library providing wrapper functions, or one may use the `--wrap` flag at link time to use the linker support for wrapper functions.

Run-time wrapping: `LD_PRELOAD`

Shared libraries are code objects that may be loaded during execution into the memory space associated with a process. Shared libraries allow considerably smaller executables, as each executable no longer has to contain its own copy of every required library. Memory may also be saved at run-time, as the operating system can share a single copy of a shared library between multiple processes. As an example, under Linux, the system C library is generally linked as a shared library.

In addition to shared libraries loaded by processes themselves, one may also force a process to load a shared library using the `LD_PRELOAD` environment variable, which instructs the loader (`ld.so`) to load the specified shared libraries. Symbol definitions in `LD_PRELOAD`ed libraries will be found before definitions in non-`LD_PRELOAD`ed libraries, so if an `LD_PRELOAD`ed library contains a definition of, say, `fopen`, then that code will be executed when the process references `fopen`. In particular, this means that we can package our wrapper functions into a shared library (`.so` file) and use `LD_PRELOAD` to cause them to be loaded.

This is all well and good; however, it provides no way for the wrapper function in our shared library to call the original version of the function it wraps. To continue our example, the version of `fopen` that we wrote would have to implement not only our own wrapping code, but also the entire functionality of `fopen`! Clearly, this is not acceptable, but there is a solution: the `dlsym` function [4]. The function `dlsym(void* handle, char* symbol)` returns the address of the symbol given as its second argument, searching the dynamic library accessible through the handle given as its first argument. Normally, handles are obtained from the `dlopen` function, which takes as an argument the name of the shared library file from which to read. However, on certain platforms (including Solaris and Linux), `dlsym` may also take the symbol `RTLD_NEXT` as a handle. When the `RTLD_NEXT` pseudo-handle is supplied, `dlsym` will return the address of the desired symbol, but instead of looking in a specific shared library, it will look in all the shared libraries that follow the current library in the library search path. Thus, assuming the only definitions of `fopen` are in our shared library and in the system C library, calling `dlsym(RTLD_NEXT, ''fopen'')` will return the address of the standard `fopen` function, to which we may assign a function pointer and later call. Thus, our wrapper functions can truly wrap system functions, as opposed to reimplementing them wholesale. Platforms not supporting `RTLD_NEXT` cannot use this method of function replacement.

The advantage of using `LD_PRELOAD` to provide wrapper functions is that it can work without modification to existing dynamically-linked binaries. The disadvantage is that the shared library providing the wrapper functions needs to be present at runtime, and this increases administrative overhead. Additionally, while it should be possible to mix C++ wrappers with C binaries [6], we found this process to be unreliable.

The `LD_PRELOAD` technique is used by the KDE project to provide applications with a legacy interface to the sound card [5], and their code was a major source of inspiration. See Appendix A-1 for an example of the `LD_PRELOAD`/shared library technique.

Link-time wrapping: the `--wrap` flag

While `dlsym` and `RTLD_NEXT` are intended to provide the ability to wrap functions at run-time, the `--wrap` argument of the GNU linker (ld) is intended to support wrapping functions at link-time. If the linker is passed the argument "`--wrap foo`", then it will resolve references to the symbol `foo` to `__wrap_foo`. It will also create a new symbol `__real_foo`, which will be resolved to the original `foo`. For example, by defining a function `__wrap_fopen` that implements whatever wrapping logic we desire and accesses the real `fopen` call through the `__real_foo symbol`, we can give the linker the argument "`--wrap fopen`" and easily create a working wrapper.

Here, replacement functions are most conveniently placed in a static library (`.a`). The advantage is that they are linked with the rest of the program, so no knowledge of the wrapping is required at run time. The downside is that this method requires relinking. Additionally, if the wrapper functions contain C++ code, then the final linking must be done by a C++ compiler.

We chose to use this method for our project, as we had access to source code (as required for relinking), and relinking was judged to be less difficult than distributing an additional shared library file. As stated earlier, we also found the `LD_PRELOAD` method to be somewhat unreliable when using C++ wrappers on a C binary (which was required for our project).

See Appendix A-2 for an example of the link-time function wrapping technique.

Windows

Under Windows, neither the `LD_PRELOAD` nor the `--wrap` flags exist per se. We chose instead to use the Detours package from Microsoft Research [3], which is specifically designed to allow the interception and processing of function calls in a minimally-intrusive manner. Detours provides two key functionalities that we use here.

First, Detours provides a program, `setdll.exe` that can modify the import section of a Win32 binary to have it load arbitrary DLLs (Dynamic Link Libraries). In particular, given wrapper functions packaged as a DLL, `setdll.exe` can be used to modify programs so that they will automatically load the wrapper DLL. In this way, Detours is similar in function to `LD_PRELOAD`.

Second, Detours provides the ability to modify the binary code of an application in memory at runtime so that a wrapper function will be called in the place of a preexisting function. It rewrites the function in question to execute an unconditional jump to a wrapper function, which may execute arbitrary code before and after the execution of the real function. The real function is made available to the wrapper function through a "trampoline" function, which executes the first few instructions removed from the real function (to make room for the unconditional jump to the wrapper) and then jumps to the real function.

The simplest way to detour a function is to use the `DETOUR_TRAMPOLINE` macro and the `DetourFunctionWithTrampoline` function. First, one calls `DETOUR_TRAMPOLINE(`*trampoline-prototype*`,` *real-function*`)`. This creates a trampoline function that will access the real function. Next, one calls `DetourFunctionWithTrampoline(`*trampoline*`,` *wrapper-function*), where *wrapper-function* is user-supplied wrapping code. This modifies the original function to call the wrapper code on invocation, thus completing the detour.

Note that when using Detours to wrap system functions, it is necessary that both the application and the DLL containing the wrapping functions be set to use MFC (Microsoft Foundation Class library) as a shared DLL. Otherwise, the wrapper functions will just end up detouring their own private copies of the system functions.

Finally, this technique will not work under Windows 95/98, unless the program is running under a debugger; this is a limitation of Detours. It will work under Windows NT/2000/XP [3].

See Appendix A-3 for an example of the Detours technique.

Macintosh OS X

Although it uses GCC, Macintosh OS X does not use the GNU linker, and thus the `--wrap` flag is unavailable to create wrapper functions. Instead, we package wrapper functions into a shared library (`.dylib`) and link that library with the application in question. The wrapper functions use the `dlsym` function and the `RTLD_NEXT` pseudo-handle to locate the functions they wrap. For this method, if the wrapper functions use C++ code, then a C++ compiler must be used to do the final linking.

Macintosh OS X does provide the `DYLD_PRELOAD` environment variable, which is analogous to `LD_PRELOAD` on systems with the GNU linker. However, `DYLD_PRELOAD` requires that the `DYLD_FORCE_FLAT_NAMESPACE` environment variable be set, and that variable breaks the `dlsym` function. This explains our use of a link-time solution rather than a run-time solution on this platform. We use a shared library rather than a static library because the `RTLD_NEXT` pseudo-handle is available only in shared libraries.

Finally, note that in order to ensure that the initialization code of dynamically-shared libraries is run, the environment variable `DYLD_BIND_AT_LAUNCH` should be set to `YES` when the program is run.

For an example of this technique, see Appendix A-4.

Possible Complications

There are a few possible complications that are common across all techniques of function wrapping. First, the GNU C library (and possibly others) uses a complex set of symbolic constants (via `#define`) to determine the exact identity of various calls. For example, if 64-bit file support is enabled, then `open64`, and not `open` will be called, even though the code contains references only to `open`. Thus, if one fails to provide a replacement for `open64`, `open` will not be wrapped when 64-bit file support is enabled.

Second, C++ allows function in-lining, which can defeat all methods of function wrapping, as it eliminates the function call itself. Under g++, the `-fno-inline` argument can be given to disable function in-lining. Some functions from the GNU C library (`stat` and `lstat`, for example) seem to be unaffected by `-fno-inline`, however.

Conclusions

We have presented techniques for intercepting function calls on the UNIX, Macintosh OS X, and Windows platforms, and we have successfully used these techniques to run bioinformatics applications under the BOINC distributed computing framework.

Acknowledgments

We thank Galen Hunt (Microsoft Research) for his assistance with Detours and Ian Taylor (Wasabi Systems) for his insight into the GNU linker. We also thank Michael Cummings for his encouragement and helpful suggestions on draft versions of this document.

References

[1] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home: An experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, November 2002. See also http://setiathome.berkeley.edu.

[2] Berkeley Open Infrastucture for Network Computing (BOINC). http://boinc.berkeley.edu/.

[3] G. Hunt and D. Brubacher. Detours: Binary interception of Win32 functions. In *Proceedings of the 3rd USENIX Windows NT Symposium*, pages 135–143, Seattle, WA, 1999. USENIX.

[4] IEEE and The Open Group. *The Open Group Base Specifications Issue 6. IEEE Std 1003.1, 2003 Edition.* The IEEE and The Open Group, 2003.

[5] The KDE Project. http://www.kde.org.

[6] I. Taylor. Personal communication, 2004.

APPENDICES

A-1 Run-time function wrapping

Here, we demonstrate the `LD_PRELOAD` run-time function replacement technique by providing a wrapper for the `fopen` function. This example was tested on RedHat version 9.

The full definition of the `fopen` function is `FILE *fopen(const char *path, const char *mode);` it opens a new file stream on the given path.

First, we need to write the wrapper function, which we place in `fopenwrap.c`. Note that under Linux, we need to `#define _GNU_SOURCE` to have access to the `RTLD_NEXT` pseudo-handle.

```
#define _GNU_SOURCE
#include <stdio.h>
#include <dlfcn.h>

FILE* fopen(const char* path, const char* mode) {
        printf("Opening %s\n", path);
        FILE* (*real_fopen)(const char*, const char*) =
                dlsym(RTLD_NEXT, "fopen");
        return real_fopen(path, mode);
}
```

Next, we compile the wrapper into a shared library.

```
gcc -fPIC -rdynamic -c fopenwrap.c
gcc -shared -o libfopenwrap.so fopenwrap.o -lc -ldl
```

Now, we create the file `fopen.c`, containing test code for our wrapper.

```
#include <stdio.h>

int main(int argc, char** argv) {
    char buf[80];
    FILE* fileptr = fopen(argv[1], "r");

    do {
        fgets(buf, 80, fileptr);
        if (feof(fileptr)) {
            break;
        }
        printf("%s", buf);
    } while(1);

    fclose(fileptr);
    return 0;
}
```

Finally, we compile the test code and run it with our wrapper library. `LD_LIBRARY_PATH` tells the loader to look for shared libraries in the current directory.

```
gcc fopen.c -o fopentest
LD_LIBRARY_PATH=. LD_PRELOAD=libfopenwrap.so ./fopentest fopen.c
```

This will print "Opening fopen.c" followed by the source of the test program.

A-2 Link-time function wrapping

We again provide a wrapping function for `fopen`, except this time we use the link-time wrapping method. This example was tested on RedHat version 9.

First, we need a C file (`fopenwrap.c`) containing the wrapper for `fopen`:

```
#include <stdio.h>

FILE* __wrap_fopen(const char* path, const char* mode) {
        printf("Opening %s\n", path);
        return __real_fopen(path, mode);
}
```

We compile it to a static library:

```
gcc -c fopenwrap.c
ar rcs libfopenwrap.a fopenwrap.o
```

We will use the same test code as in the previous example, but we will compile it differently: Note how we use the `-Xlinker` option to `gcc` to pass arguments through to the linker.

```
gcc fopen.c -o fopentest -L. -lfopenwrap -Xlinker --wrap -Xlinker fopen
```

Running the test program with a readable file as its only argument will print the line "Opening <file>" followed by the contents of the file.

A-3 Windows function wrapping using Detours

In this example, we use Microsoft Detours to wrap `fopen` yet again. This example was tested using Visual C++ from Visual Studio .NET 2003.

First, we need to create the DLL that will provide our wrapper functions. In VC++, create a new project of type "Win32 console project." In the second screen of the wizard, select the "Application Settings" tab and configure a DLL with the "Export symbols" box checked.

After finishing the wizard, open the properties screen for the new project. At the top of the dialog box, change the configuration from "Active(Debug)" to "All Configurations." Next, in the "General" configuration properties section, change "Use of MFC" from "Use Standard Windows Libraries" to "Use MFC in a Shared DLL." Finally, copy `detours.lib` and `detours.h` from the Detours distribution to the project directory.

The code for the `fopen` function is below. The contents of this listing would go in the primary `.cpp` file of the DLL.

```
#include "stdafx.h"
#include <stdio.h>
#include "detours.h"
#pragma warning(disable:4100)

DETOUR_TRAMPOLINE(FILE* TrampFopen(CONST CHAR* filename, CONST CHAR* mode), fopen);
FILE* WrapFopen(CONST CHAR* filename, CONST CHAR* mode) {
    printf("Opening %s\n", filename);
    return TrampFopen(filename, mode);
}

BOOL APIENTRY DllMain( HANDLE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                     )
{
    printf("DLLMain\n");
    switch (ul_reason_for_call)
    {
        case DLL_PROCESS_ATTACH:
            DetourFunctionWithTrampoline((PBYTE)TrampFopen, (PBYTE)WrapFopen);
            break;
        case DLL_THREAD_ATTACH:
            break;
        case DLL_THREAD_DETACH:
            break;
        case DLL_PROCESS_DETACH:
            DetourRemove((PBYTE)TrampFopen, (PBYTE)WrapFopen);
            break;
    }
    return TRUE;
}
```

At this point, we can go ahead and build the DLL.

Now, we need test code for the DLL. Create a new VC++ project, choosing Win32 console application as the type and accepting all the default options. As before, change the “Use of MFC” option to “Use MFC in a shared DLL.” Finally, use the test code used in the prior two sections as the code for the application, and compile it.

To test the driver code, we need to gather the following files in one directory: `setdll.exe` (from the Detours distribution), the DLL containing the `fopen` wrapper, and the test code. Assuming the DLL is called `fopenwrap.dll` and the test code is called `fopen.exe`, we use the following commands to bind the DLL to the test code and run the test:

```
setdll.exe -d:fopenwrap.dll fopen.exe
fopen c:\autoexec.bat
```

A-4 Macintosh OS X function wrapping

Once again, we wrap the `fopen` function. This example was tested on OS X 10.3.

We use the same test code as before, in the file `fopen.c`. For the wrapper, we put the following in `fopenwrap.c`:

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

FILE* fopen(const char *path, const char *mode)
{
    printf("This is a wrapper function for fopen.\n");
    FILE* (*real_fopen)(const char*, const char*) =
            (FILE* (*)(const char*, const char*)) dlsym(RTLD_NEXT, "fopen");
    return real_fopen (path, mode);
}
```

To compile the code, we issue the following two commands:

```
gcc -fno-common -c fopenwrap.c
gcc -dynamiclib -o libwrapper.dylib fopenwrap.o
```

Finally, we compile the test code and link it with the wrapper library:

```
gcc fopen.c -o fopentest -L. -lwrapper
```

Remember to set the `DYLD_BIND_AT_LAUNCH` environment variable when running the test code:

```
DYLD_BIND_AT_LAUNCH=YES ./fopentest
```