MONDO: A Shared Library and Dynamic Linking Monitor

C. Donour Sizemore, Jacob R. Lilly, and David M. Beazley
Department of Computer Science
University of Chicago
Chicago, Illinois 60637
{donour,jrlilly,beazley}@cs.uchicago.edu

March 15, 2003

Abstract

Dynamic modules are one of the most attractive features of modern scripting languages. Dynamic modules motivate programmers to build their applications as small components, then glue them together. They rely on the runtime linker to assemble components and shared libraries as the application runs. MONDO, a new debugging tool, provides programmers with the ability to monitor in real-time the dynamic linking of a program. MONDO supplies programmers with a graphical interface showing library dependencies, listing symbol bindings, and providing linking information used to uncover subtle programming errors related to the use of shared libraries. The use of MONDO requires no modification to existing code or any changes to the dynamic linker. MONDO can be used with any application that uses shared libraries.

1 Introduction

For the past 10-15 years, shared libraries and dynamic linking have changed the way application programmers write software. Instead of writing huge monolithic applications, it is quite common to execute smaller software components, extension modules, and plugins. For example, when programmers use scripting languages like Python or Tcl, they also tend to use a collection of loosely-coupled dynamically loadable extension modules. Similarly, dynamic modules are commonly used to extend web servers, browsers, and other large programming environments.

Although dynamic linking offers many benefits such as increased modularity, simplified maintenance, and extensibility, it has also produced a considerable amount of programmer confusion. Few programmers would claim to really understand how dynamic linking actually works. Moreover, the runtime linking process depends heavily on the system configuration, environment variables, subtle compiler and linker options, and the flags passed to low-level dynamic loader. The interaction of these pieces tends to produce a very obscured picture of what is actually going on inside an application that uses shared libraries. Unfortunately, traditional debugging tools are of little assistance since they are primarily concerned with errors in program logic rather than errors that arise from the way in which a program is constructed and linked.

To address this limitation, we present a special purpose debugger, MONDO (Monitor of Dynamic Objects), that is designed to help programmers discover problems related to the run-time linking of an application. MONDO works by watching real-time debug traces generated by the run-time linker (ld.so.1) and using the output information to construct a global view of an application, its libraries, and dynamically loaded modules (if any). Using this information, it is possible examine symbol bindings, library dependencies, as well as to uncover subtle programming problems related to linking that are otherwise hidden from the programmer by the limitations of classic debugging tools.

In the first part of this paper, we illustrate dynamic linking and some of the common programming errors that arise. Next, we describe how runtime linking information can be extracted from the loader. This information is used to construct the MONDO debugger and how it can be used by programmers.

2 An Overview of Dynamic Linking

To create an executable, a "linker" is used to assemble the appropriate object files and libraries into a runnable program [2]. Linking is nothing more than the binding of symbolic names to memory addresses. For example, if you have some procedure `foo()` in your application, the linker calculates where in memory `foo()` will reside and patches all references to `foo()` making sure they point to the memory address of `foo()`. Most programmers view linking as merely the final step of building a program. That is,can one simply runs the linker and it collects all of the object files and library functions into some huge "bundle of bits" that is the program. A popular misconceptions of program state is that once linked the program runs statically without any dynamic changes (i.e., one just executes the program).

In reality, the situation is much more complicated and convoluted than the previously described model. Most modern systems use dynamic linking–a technique in which much of the actual linking is deferred until runtime. For example, when you create a program like this,

```
% ld $(OBJS) -lsocket -lthread -ldl
```

The linker merely records the names of libraries in the executable instead of linking them. A command such as `ldd` can be used to list these library and their dependencies. For example:

```
% ldd a.out
libsocket.so.1 => /usr/lib/libsocket.so.1
libthread.so.1 => /usr/lib/libthread.so.1
libdl.so.1 => /usr/lib/libdl.so.1
libc.so.1 => /usr/lib/libc.so.1
```

When a dynamically linked program runs, control is first passed to a special runtime loader (often called `ld.so.1`). The loader is then responsible for locating, loading, and binding library dependencies to the the executable–a process that occurs at runtime.

To improve performance, most of the runtime binding occurs lazily. Library symbols are not actually bound until they are used for the first time by a program. This binding is managed by routing procedure calls through a special procedure linking table (PLT), a jump table use to match procedures with memory addresses. Initially, the PLT is set up to pass control back to the dynamic loader when a procedure is called for the first time. When this occurs, the dynamic linker locates the symbol in a library and patches the PLT to point to the newly bound symbol. Many programmers are unaware that linking occurs throughout the execution of a program. For instance, when a user selects certain application features, the linker quietly runs underneath the program–binding all of the newly used symbols as they are needed.

In addition to automatically binding libraries, the runtime loader can also be used to explicitly load new program modules through a special dynamic loading API. Using calls such as `dlopen()` and `dysym()`, programmers can load shared object files, search for symbols, and invoke functions. This API is the basis for systems that support dynamically loadable extension modules and plugins.

3 Enter MONDO

Unlike in conventional programming langauges (C, C++, etc), users routinely change the "structure" of their program when they import modules. In scripting languages errors in program construction can be considered programming errors. A single module may introduce many libraries with no clear way of watching what is happening to a process as they are linked. Here a mistake in program logic results in an incorrectly built program.

The runtime linker (ld.so.1) on many systems can supply real-time updates as it binds symbols from dynamic objects. The LD_DEBUG environment variable allows the user to collect information about libraries, dependencies, bindings, and relocations as they are processed by ld.so.1 during program execution. Figure 1 illustrates some of this output for Solaris.

The `ld.so.1` debugging trace contains a wealth of information about almost everything the runtime linker is doing during program execution. Moreover, this information is presented in real-time–making it qualitatively different than the information provided by static library diagnostic tools such as `ldd`, `nm`, or `elfdump`. However, the debug trace is often unreadable and contains too much unordered information to be easily usable by programmers.

MONDO is a tool that synthesizes real-time data from the dynamic loader with the information that is available from existing library diagnostic tools (e.g., `nm`, `ldd`, etc.). Moreover, it tries to present this information to the user in a coherent manner using an interactive graphical user interface. The system consists of two components; a real-time data collector that extracts data from the `ld.so.1` trace and an analysis tool that assembles this data, combines it with static library information, and presents it to the user.

To use MONDO, a programmer simply types "mondo" followed by a normal UNIX shell command. For example:

```
% mondo a.out
```

This sets up the proper environment variables for monitoring, launches the debugger, and executes the requested command. From this point forward, the user is able to monitor the library bindings of the command.

4 Data Collection

As previously mentioned, MONDO collects information from `ld.so.1` using its debugging mode. This is enabled by setting the LD_DEBUG and LD_DEBUG_OUTPUT environment variables of a process. Debugging data is redirected to a file where it is read and parsed by MONDO as the process executes. Information of interest includes library search paths, library loading, library dependencies, symbol bindings, and relocations. In addition to collecting the trace data, MONDO also collects information from the library files themselves. This is used to build a model of what libraries have been loaded, what symbols are contained in those libraries, and which symbols have actually been bound by the runtime linker.

MONDO can simultaneously monitor any number of programs and is limited only by the underlying operating system (number of process, open file handles, etc.). Since tracing is enabled by the environment (and environments are preserved after calls to exec()), any process spawned by a traced program will be traced itself. Process ID is provided inline with traces data (first column in fig.1) and `ld.so.1` is able to dump trace data to a file corresponding the PID. MONDO can follow the growth of a process tree without the need to do system call tracing.

Function overloading, Namespaces, etc. do not exists at the object level. To support these features C++ compiles mangle type names. The mangled names reflect typing and scope information. MONDO uses the demangingle code from GCC to demangle these symbols names.

5 MONDO Interface

MONDO presents the loaded libraries and symbols as a forest. The root of the each tree is a traced process. Below each process is a list of every library that process has opened. Figure 2 shows MONDO using the GTK+ 2.2.0 libraries. Under each library is a list symbols and where they were bound. Color indicates the type of each symbol. Features include:

- Trace new program

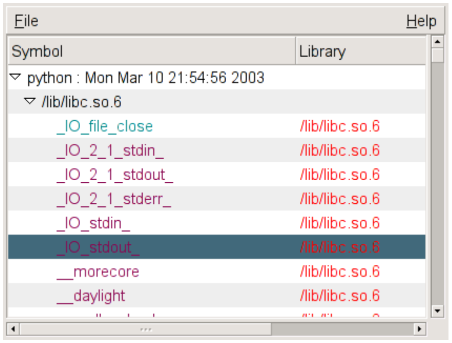
```
% env LD_DEBUG=basic,bindings,files,libs,symbols,detail ls
...
15165: file=ls;  analyzing  [ RTLD_LAZY  RTLD_GLOBAL  RTLD_WORLD  RTLD_NODELETE ]
15165:     permit:       UNUSED  [ GLOBAL  NODELETE ]
15165:
15165: file=libc.so.1;  needed by ls
15165:
15165: find object=libc.so.1; searching
15165:  search path=/usr/local/lib  (configuration default - /var/ld/ld.config)
15165:  search path=/usr/lib  (configuration default - /var/ld/ld.config)
15165:  trying path=/usr/local/lib/libc.so.1
15165:  trying path=/usr/lib/libc.so.1
15165: file=/usr/lib/libc.so.1  [ ELF ]; generating link map
...
15165: symbol=malloc;  lookup in file=ls  [ ELF ]
15165: symbol=malloc;  lookup in file=/usr/lib/libc.so.1  [ ELF ]
15165: symbol=malloc;  lookup in file=/usr/platform/SUNW,Ultra-80/lib/libc_psr.so.1  [ ELF ]
15165: binding file=ls (0x11704:0x1704) at plt[11]:full to
       file=/usr/lib/libc.so.1 (0xff2c1880:0x41880): symbol ‘malloc’
15165: symbol=lstat64;  lookup in file=ls  [ ELF ]
15165: symbol=lstat64;  lookup in file=/usr/lib/libc.so.1  [ ELF ]
15165: symbol=lstat64;  lookup in file=/usr/platform/SUNW,Ultra-80/lib/libc_psr.so.1  [ ELF ]
15165: binding file=ls (0x12d38:0x2d38) at plt[36]:full to
       file=/usr/lib/libc.so.1 (0xff318358:0x98358): symbol ‘lstat64’
...
```

Figure 1: Sample LD_DEBUG output

- View debug traces
- Generate library dependence graphs
- Disassembling functions
- Save/Load traced sessions

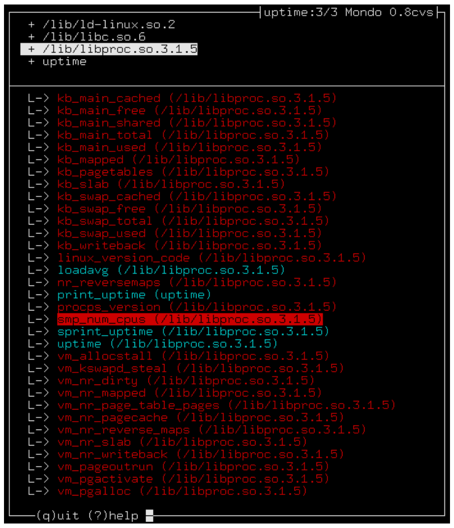
MONDO provides both a X11 and a terminal interface. If X11 is present, and the GTK2 [5] libraries are installed, MONDO provides a threaded, windowed interface, designed to be easy and intuitive for most X11 users. The interface is deliberately simple to avoid distracting the user from the programs that are being traced.

If X11 is not available MONDO provides the user with a terminal interface built around the `curses` module. Curses is enabled by the `-curses` flag on the command line or if MONDO fails to connect to an X server. Curses may also be preferred for remote debugging across a high latency connection such as a modem or satellite. It provides nearly identical functionality to the X11 interface, although users are unable to view more than a single trace at a time.



Symbol	Library
python : Mon Mar 10 21:54:56 2003	
/lib/libc.so.6	
_IO_file_close	/lib/libc.so.6
_IO_2_1_stdin_	/lib/libc.so.6
_IO_2_1_stdout_	/lib/libc.so.6
_IO_2_1_stderr_	/lib/libc.so.6
_IO_stdin_	/lib/libc.so.6
_IO_stdout_	/lib/libc.so.6
__morecore	/lib/libc.so.6
__daylight	/lib/libc.so.6

Figure 2: MONDO via GTK2



```
                                   uptime:3/3 Mondo 0.8cvs
 + /lib/ld-linux.so.2
 + /lib/libc.so.6
 + /lib/libproc.so.3.1.5
 + uptime

 L-> kb_main_cached (/lib/libproc.so.3.1.5)
 L-> kb_main_free (/lib/libproc.so.3.1.5)
 L-> kb_main_shared (/lib/libproc.so.3.1.5)
 L-> kb_main_total (/lib/libproc.so.3.1.5)
 L-> kb_main_used (/lib/libproc.so.3.1.5)
 L-> kb_mapped (/lib/libproc.so.3.1.5)
 L-> kb_pagetables (/lib/libproc.so.3.1.5)
 L-> kb_slab (/lib/libproc.so.3.1.5)
 L-> kb_swap_cached (/lib/libproc.so.3.1.5)
 L-> kb_swap_free (/lib/libproc.so.3.1.5)
 L-> kb_swap_total (/lib/libproc.so.3.1.5)
 L-> kb_swap_used (/lib/libproc.so.3.1.5)
 L-> kb_writeback (/lib/libproc.so.3.1.5)
 L-> linux_version_code (/lib/libproc.so.3.1.5)
 L-> loadavg (/lib/libproc.so.3.1.5)
 L-> nr_reversemaps (/lib/libproc.so.3.1.5)
 L-> print_uptime (uptime)
 L-> procps_version (/lib/libproc.so.3.1.5)
 L-> smp_num_cpus (/lib/libproc.so.3.1.5)
 L-> sprint_uptime (/lib/libproc.so.3.1.5)
 L-> uptime (/lib/libproc.so.3.1.5)
 L-> vm_allocstall (/lib/libproc.so.3.1.5)
 L-> vm_kswapd_steal (/lib/libproc.so.3.1.5)
 L-> vm_nr_dirty (/lib/libproc.so.3.1.5)
 L-> vm_nr_mapped (/lib/libproc.so.3.1.5)
 L-> vm_nr_page_table_pages (/lib/libproc.so.3.1.5)
 L-> vm_nr_pagecache (/lib/libproc.so.3.1.5)
 L-> vm_nr_reverse_maps (/lib/libproc.so.3.1.5)
 L-> vm_nr_slab (/lib/libproc.so.3.1.5)
 L-> vm_nr_writeback (/lib/libproc.so.3.1.5)
 L-> vm_pageoutrun (/lib/libproc.so.3.1.5)
 L-> vm_pgactivate (/lib/libproc.so.3.1.5)
 L-> vm_pgalloc (/lib/libproc.so.3.1.5)

 (q)uit (?)help
```

Figure 3: MONDO via curses

If the Graphviz [1] package is available, the user can generate a library dependence graph (Fig. 4) at arbitrary points of program execution. All of Graphviz's output formats are supported (both vector and bitmap image formats). Sometime a library dependence is reinforced by several thousand bindings to a single file. The number of bindings is not currently reflected in dependence graphs. Graphviz can also remove excess cycles from the graph and concatence edges to make the graphs more readable.

6 Implementation

MONDO currently consists of 3K lines of Python [3] and 15K lines of C for supporting functions (c++ demangling, elf processing). Parsing of the debug trace is easily accomplished using regular expressions. However, work is currently underway to rewrite the parser using PLY [6], an SLR(1) parser.

Internally, the MONDO `session` object stores all data related to a program trace: program name, argument, binding history. The objects can be saved to disk and loaded later for visualization of comparison against later sessions. Figure 5 shows MONDO's internal components.

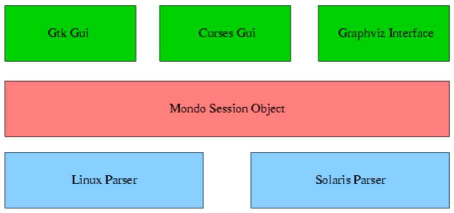


Figure 5: MONDO's Functional Units

The graphical user interface is being implemented with the PyGTK bindings to GTK2, and ncurses. Python has allowed for very rapid developement while maintaining equivalent functionality across several interfaces. MONDO also hooks other system tools, such as 'elfdump' and 'objdump', to obtain detailed symbol data.

6.1 Supported Platorms

MONDO can trace programs on Solaris 2.x and Linux 2.x with Glibc. It requires Python 2, a recent version of SWIG (1.3.17 or greater), and either X11 or curses.

There exists no standard for the debugging output of the run-time linker. The Solaris linker provides basic documentation on the features available. Although no grammar for the output is given, most binding information can easily be extracted from the context. Solaris also provides details about where symbols are represented in the PLT. The source to ld-linux.so, Linux's realtime linker, is freely available to the public. This makes it easier to determine exactly when and where output will be generated. It is necessary for MONDO to maintain seperate parsers for each platform, but all other subsystems are platform independent. The exceptions are a few special pieces of data; the PID and the /proc interface.

7 Limitations

It is unlikely that the implementers of `ld.so.1` intended the debugging trace to serve as the input to

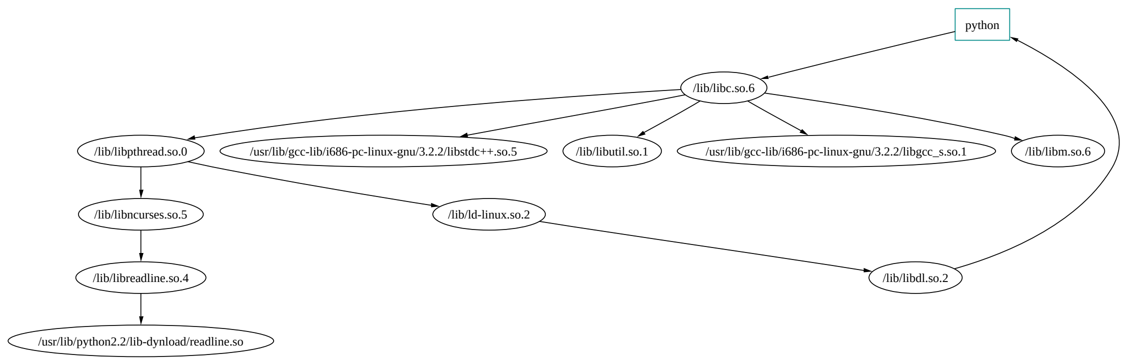

python
/lib/libc.so.6
/lib/libpthread.so.0
/usr/lib/gcc-lib/i686-pc-linux-gnu/3.2.2/libstdc++.so.5
/lib/libutil.so.1
/usr/lib/gcc-lib/i686-pc-linux-gnu/3.2.2/libgcc_s.so.1
/lib/libm.so.6
/lib/libncurses.so.5
/lib/ld-linux.so.2
/lib/libreadline.so.4
/lib/libdl.so.2
/usr/lib/python2.2/lib-dynload/readline.so

Figure 4: Python Library Dependence Graph

other programs. The output is relatively unstructured and it's not standardized across platforms. The level of detail available in the trace also varies across platforms.

It's natural to want to extend the software to work on other platforms; The BSD variations and Mac OSX come to mind. Preliminary tests show that these platforms do not provide traces through the `LD_DEBUG` mechanism. No tracing is possible on Windows as it has a different run-time linking module.

The generation of trace data adds a noticable performance impact to program startup and module loading. As symbols are bound, a large amount of trace data may be generated. However, this data is only generated once for each bound symbol. Therefore, a traced application generally incurs no performance penalty once it has bound the most commonly used symbols.

Finally, Python hasn't been particularly fast. The debug data is often several megabytes in size and large applications may involve tens of thousands of symbols and dozens of libraries. As a result, it can take quite a while to load all of this information and to launch the monitor. So far, the focus has been interesting functionality. Optimizations could give a large speed boost.

8 Related Work

We are not aware of any tools quite like MONDO. Existing Unix commands like `ldd` and `nm` provide information about libraries, but this information is static and presented in isolation. Profiling tools like `prof` tell a programmer which procedures are used, but require special recompilation. Moreover, a profile doesn't contain library information. A traditional debugger provides information about loaded libraries, but doesn't provide the same kind of runtime information as MONDO since the debugger doesn't tend to interact with the runtime loader. Component frameworks such as COM often provide tools for inspecting the contents of component modules . However, these are too specialized to be used with shared libraries in general.

9 Applications

Previously, information about runtime linking and symbol binding has not been provided to the developer in any cohesive manner. Tools such as `ldd` and `elfdump` merely provide a static snapshot of a library without runtime binding information. We think that the primary utility of MONDO is its ability to show runtime linking information. This may be useful in debugging obscure shared library problems or for merely gaining a better understanding of how shared libraries work. This information would also be of particular interest to developers who have to work with extension modules and plugins.

MONDO could also be a useful software engineering tool. By being able to view the structure of an application, a programmer might be able to discover better ways of organizing a program or they could identify unused parts of an application.

10 Future Work

It would be useful to be able stop running process and alter their state like tradition debuggers. If single step execution were available, MONDO can use it's history of bindings to construct a symbol timeline. These timeline could be used to influence future executions of the same program.

Since MONDO has intimate knownledge of a programs construction, it could relink a program while it is running. A users could potentially cook up a piece of code in C (or Python, Perl, etc.), compile it, then link it into a running application; overwriting an existing symbol. For example, a user could replace libc's version of malloc with one that gave debugging output or one optimized for performance. Taking this one step further, one could embed a just-in-time compiler in MONDO and provide an interface that allows the user to input code and the symbol they want that code chunck bound to.

10.1 Prelinking

Libary prelinking is becoming popular as a way to decrease application load time. Application such as OpenOffice commonly have over 100,000 symbol bindings during initialization. Users prelink these symbols to avoid having to bind them every time they run their application. The resulting program is still a dynamic executable, but it does not have to load dependcies externally. Optionally, a user can choose to only prelink certain libraries; the ones that would provide the greatest speed benefit. To do this, the user needs the quanatative data on library bindings that MONDO can provide. Prelinking is possible with recent version of GNU Libc.

Another possiblity would be to use Emacs' method for unexec. The Emacs distribution includes code to dump an in memory process image out to an executable. MONDO could manually prelink by loading necessary symbols, then calling unexec to generate a new program.

11 Status

MONDO is currently a work in progress and in version 0.9 with the 1.0 release expected shortly.

Details are available at: `http://systems.cs.uchicago.edu/mondo`.

12 Acknowledgements

We would like to thank the following people who have, either directly or through the community, comtributed to our project: Sam TH, the Graphviz Team, Stefan Jones, Lars Wirzenius, and Jon Riehl.

References

[1] E.R. Gansner and S.C North, *An open graph visualization system and its applications to software engineering*, Softw, Pract. Exper., 00(S1),1-5(1999).

[2] J. Levine, *Linkers & Loaders*, Morgan Kaufmann Publishers, (2000).

[3] M. Lutz, *Programming Python*, O'Reilly & Associates, (1996).

[4] J. Grayson, *Python and Tkinter Programming*, Manning, (2000).

[5] GTK+ Team, *Gimp Toolkit*, `http://www.gtk.org`

[6] David M. Beazley, *Python Lex and YACC*, `http://systems.cs.uchicago.edu/ply`