Assignment 6. Dynamic linking

Useful pointers

- M. Tim Jones, Anatomy of Linux dynamic libraries. IBM developerWorks (2008).
- David A. Wheeler, Program Library HOWTO 1.20 (2003).
- vDSO – overview of the virtual ELF dynamic shared object (2019).
- libffi – A Portable Foreign Function Interface Library (2014).

Laboratory: Who's linked to what?

As usual, keep a log in the file `lab.txt` of what you do in the lab so that you can reproduce the results later. This should not merely be a transcript of what you typed: it should be more like a true lab notebook, in which you briefly note down what you did and what happened.

For this laboratory, you will find out about which programs are linked to which libraries.

1. Compile, build and run the program simpgmp.c in C on the SEASnet GNU/Linux servers. Since it uses GMP you will need to use the linker flag `-lgmp` to build this program.
2. Use the program to compute 2**24 (i.e., 2^{24}, or 2 to the 24th power) and 2**(2**24); verify that the latter number has 5,050,446 decimal digits, starts with "1818" and ends with "7536".
3. Use the `ldd` command to see which dynamic libraries your simple program uses.
4. Use the `strace` command to see which system calls your simple program makes. Which of these calls are related to dynamic linking and what is the relationship?
5. Suppose your student ID is the 9-digit number *nnnnnnnnn*. On a SEASnet GNU/Linux server, run the shell command "`ls /usr/bin | awk '(NR-`*nnnnnnnnn*`)%251 == 0'`" to get a list of nine or so commands to investigate.
6. Invoke `ldd` on each command in your list. If there are error messages, investigate why they're occurring.
7. Get a sorted list of every dynamic library that is used by any of the commands on your list (omitting duplicates from your list).

Homework: Split an application into dynamically linked modules

In this homework you will divide a small example application into dynamically linked modules and a main program, so that the resulting executable does not need to load code that it doesn't need. Although this is just a toy example which would probably not be worth optimizing in this way, in real life many applications use dynamic linking to improve performance in common cases, and the skills used in this small exercise can be helpful in larger programs.

The skeleton tarball contains the following:

- A file `randall.c` that is a single main program, which you are going to split apart.
- A `Makefile` that builds the program `randall`.
- Two files `randcpuid.h` and `randlib.h` that specify two interfaces for libraries that you need to implement when you split `randall.c` apart.

First, read and understand the code in `randall.c`. Do not modify it, or any of the other files in the skeleton tarball.

Second, split the `randall` implementation by copying its source code into the following modules, which you will need to modify to get everything to work:

1. `randcpuid.c` should contain the code that determines whether the current CPU has the RDRAND instruction. It should start by including `randcpuid.h` and should implement the interface described by `randcpuid.h`.
2. `randlibhw.c` should contain the hardware implementation of the random number generator. It should start by including `randlib.h` and should implement the interface described by `randlib.h`.
3. `randlibsw.c` should contain the software implementation of the random number generator. Like `randlibhw.c`, it should start by including `randlib.h` and should implement the interface described by `randlib.h`. Since the software implementation needs initialization and finalization, this implementation should also define an initializer and a finalizer function, using GCC's "`__attribute__ ((constructor))`" and "`__attribute__ ((destructor))`" declaration specifiers.
4. `randmain.c` should contain the main program that glues together everything else. It should include `randcpuid.h` (as the corresponding module should be linked statically) but not `randlib.h` (as the corresponding module should be linked after `main` starts up). Depending on whether `randcpuid` reports that the hardware supports the RDRAND instruction, this main program should dynamically link the hardware-oriented or software-oriented implementation of `randlib`, doing the dynamic linking via `dlopen` and `dlsym`. Also, the main program should call `dlclose` to clean up before exiting. Like `randall`, if any function called by the main program fails, the main program should report an error and exit with nonzero status.

Each module should include the minimal number of include files; for example, since `randcpuid.c` doesn't need to do I/O, it shouldn't include `stdio.h`. Also, each module should keep as many symbols private as it can; for example, since `randcpuid` does not need to export the `cpuid` function, that function should be `static` and not `extern`.

Next, write a makefile include file `randmain.mk` that builds the program `randmain` using three types of linking. First, it should use static linking to combine `randmain.o` and `randcpuid.o` into a single program executable `randmain`. Second, it should use dynamic linking as usual to link the C library and any other necessary system-supplied files before its `main` function is called. Third, after `main` is called, it should use dynamic linking via `dlsym` as described above. `randmain.mk` should link `randmain` with the options "`-ldl -Wl,-rpath=$PWD`". It should compile `randlibhw.c` and `randlibsw.c` with the `-fPIC` options as well as the other GCC options already used. And it should build shared object files `randlibhw.so` and `randlibsw.so` by linking the corresponding object modules with the `-shared` option, e.g., "`gcc ... -shared randlibsw.o -o randlibsw.so`".

The supplied `Makefile` includes `randmain.mk`, so you should be able to type just `make` to build all four files: `randall`, `randmain`, `randlibhw.so`, and `randlibsw.so`. If `randmain` needs to generate any random numbers, it loads either `randlibhw.so` or `randlibsw.so` (but not both) to do its work. You can verify this by using "`strace ./randmain`" or by using a debugger.

Submit

Submit the file `dlsubmission.tgz`, which you can build by running the command `make submission`.

All files should be ASCII text files, with no carriage returns, and with no more than 200 columns per line. The shell command

```
expand lab.txt randmain.mk \
  randcpuid.c randlibhw.c randlibsw.c randmain.c |
  awk '/\r/ || 200 < length'
```

should output nothing.

$Id: assign6.html,v 1.25 2019/11/05 21:08:48 eggert Exp $