Building cross compilers

Linux as the target platform

Obtained the source code to gcc, one can just follow the instructions given in the INSTALL file for GCC.

configure --target=i486-linux --host=XXX
on platform XXX followed by a make should do the trick.

Note that you will need the Linux includes, the kernel includes, and also to build the cross assembler and cross linker from the sources in .

RPM files

Readymade binary files can be installed directly.
Rpm -ivh <package>

PROGRAM LIBRARY

A ``program library'' is simply a file containing compiled code (and data) that is to be incorporated later into a program; program libraries allow programs to be more modular, faster to recompile, and easier to update.

Program libraries can be divided into three types:

- static libraries,
- shared libraries, and
- dynamically loaded (DL) libraries.

static libraries: which are copied into a program executable before the program can be run.

shared libraries: which are loaded at program start-up and shared between programs.

dynamically loaded (DL) libraries: which can be loaded and used at any time while a program is running.

Static Libraries

- Static libraries are simply a collection of ordinary object files;
- Conventionally, static libraries end with the ``.a'' suffix.
- created using the ar (archiver) program.
- Static libraries permit users to link to programs without having to recompile its code, saving recompilation time.
- In theory, code in static ELF libraries that is linked into an executable should run slightly faster (by 1-5%) than a shared library or a dynamically loaded library (?)
- To create a static library, or to add additional object files to an existing static library, use a command like this:

```
ar rcs my_library.a file1.o file2.o
```

One can use a static library by invoking it as part of the compilation and linking process when creating a program executable.

use the -l option to specify the library

```
gcc mathdemo.c -lm
```

Be careful about the order of the parameters when using gcc; the -l option is a linker option, and thus needs to be placed AFTER the name of the file to be compiled.

Shared Libraries

- Shared libraries are libraries that are loaded by programs when they start.
- When a shared library is installed properly, all programs that start afterwards automatically use the new shared library.
- Approach used by Linux permits you to:

  * update libraries and still support programs that want to use older, non-backward-compatible versions of those libraries;
  * override specific libraries or even specific functions in a library when executing a particular program.
  * do all this while programs are running using existing libraries.

Shared Library Conventions

- For shared libraries to support all of these desired properties, a number of conventions and guidelines must be followed.
- One needs to understand the difference between a library's names, in particular its ``soname'' and ``real name'' (and how they interact).
- One also needs to understand where they should be placed in the filesystem.
- Every shared library has a special name called the ``soname''. The soname has the prefix ``lib'', the name of the library, the phrase ``.so'', followed by a period and a version number
- Version number is incremented whenever the interface changes (as a special exception, the lowest-level C libraries don't start with ``lib'').

Shared Library Conventions

- Every shared library also has a ``real name'', which is the filename containing the actual library code.

- The real name adds to the soname a period, a minor number, another period, and the release number. (The last period and release number are optional.)

- The minor number and release number support configuration control by letting you know exactly what version(s) of the library are installed.

•The compiler uses simply the soname without any version number, when requesting a library, (``linker name'')

•Programs, when they internally list the shared libraries they need, should only list the soname they need.

•Conversely, when we create a shared library, we only create the library with a specific filename (with more detailed version information).

•When we install a new version of a library, we install it in one of a few special directories and then run the program ldconfig(8).

•ldconfig examines the existing files and creates the sonames as symbolic links to the real names, as well as setting up the cache file /etc/ld.so.cache

•ldconfig doesn't set up the linker names; typically this is done during library installation, and the linker name is simply created as a symbolic link to the ``latest'' soname or the latest real name.

•Thus, /usr/lib/libreadline.so.3 is a fully-qualified soname, which ldconfig would set to be a symbolic link to some realname like /usr/lib/libreadline.so.3.0.

•There should also be a linker name, /usr/lib/libreadline.so which could be a symbolic link referring to “/usr/lib/libreadline.so.3.”

Filesystem Placement

- The GNU standards recommend installing by default all libraries in /usr/local/lib when distributing source code (and all commands should go into /usr/local/bin).

- They also define the convention for overriding these defaults and for invoking the installation routines.

- The Filesystem Hierarchy Standard (FHS) discusses what should go where in a distribution

- According to the FHS, most libraries should be installed in “/usr/lib”, but libraries required for startup should be in /lib and libraries that are not part of the system should be in “/usr/local/lib”.

•One complication is that Red Hat-derived systems don't include /usr/local/lib by default in their search for libraries;

•Other standard library locations include /usr/X11R6/lib for X-windows.

How Libraries are Used

- On GNU glibc-based systems, including all Linux systems, starting up an ELF binary executable automatically causes the program loader to be loaded and run.
- On Linux systems, this loader is named /lib/ld-linux.so.X (where X is a version number).
- This loader, in turn, finds and loads all other shared libraries used by the program.
- The list of directories to be searched is stored in the file “/etc/ld.so.conf”.
- Many Red Hat-derived distributions don't normally include /usr/local/lib in the file /etc/ld.so.conf.
- Adding /usr/local/lib to /etc/ld.so.conf is a common ``fix'' required to run many programs

•Searching all of these directories at program start-up would be grossly inefficient, so a caching arrangement is actually used.

•ldconfig by default reads in the file /etc/ld.so.conf, sets up the appropriate symbolic links in the dynamic link directories and then writes a cache to /etc/ld.so.cache that's then used by other programs.

•This greatly speeds up access to libraries.

•The implication is that ldconfig must be run whenever a DLL is added, when a DLL is removed, or when the set of DLL directories changes; running ldconfig is often one of the steps performed by package managers when installing a library.

•On start-up, then, the dynamic loader actually uses the file /etc/ld.so.cache and then loads the libraries it needs.

Environment Variables

LD_LIBRARY_PATH

In Linux, the environment variable LD_LIBRARY_PATH is a colon-separated set of directories where libraries should be searched for first, before the standard set of directories; this is useful when debugging a new library or using a nonstandard library for special purposes.

Creating a Shared Library

- Creating a shared library is easy:
- First, create the object files that will go into the shared library using the gcc -fPIC or -fpic flag.
- The -fPIC and -fpic options enable ``position independent code'' generation, a requirement for shared libraries;
- Pass the soname using the -Wl gcc option.

```
gcc -shared -Wl,-soname,your_soname  -o library_name file_list library_list
```

•Here's an example, which creates two object files (a.o and b.o) and then creates a shared library that contains both of them.

•Note that this compilation includes debugging information (-g) and will generate warnings (-Wall), which aren't required for shared libraries but are recommended.

•The compilation generates object files (using -c), and includes the required -fPIC option:

```
gcc -fPIC -g -c -Wall a.c
gcc -fPIC -g -c -Wall b.c
gcc -shared -Wl,-soname,libmystuff.so.1 \
    -o libmystuff.so.1.0.1 a.o b.o -lc
```

Installing and Using a Shared Library

•The simple approach is simply to copy the library into one of the standard directories (e.g., /usr/lib) and run ldconfig(8).

•First, you'll need to create the shared libraries somewhere. Then, you'll need to set up the necessary symbolic links, in particular a link from a soname to the real name

•The simplest approach is to run:

• ldconfig -n directory_with_shared_libraries

•Finally, when you compile your programs, you'll need to tell the linker about any static and shared libraries that you're using. Use the -l and -L options for this.

You can see the list of the shared libraries used by a program using ldd(1).
So, for example, you can see the shared libraries used by ls by typing:

```
ldd /bin/ls
```

Generally you'll see a list of the sonames being depended on, along with the directory that those names resolve to.
In practically all cases you'll have at least two dependencies:

* /lib/ld-linux.so.N (where N is 1 or more, usually at least 2). This is the library that loads all other libraries.
* libc.so.N (where N is 6 or more). This is the C library. Even other languages tend to use the C library (at least to implement their own libraries), so most programs at least include this one.

Dynamically Loaded (DL) Libraries

- Dynamically loaded (DL) libraries are libraries that are loaded at times other than during the startup of a program.
- They're particularly useful for implementing plugins or modules, because they permit waiting to load the plugin until it's needed.

- In Linux, DL libraries aren't actually special from the point-of-view of their format; they are built as standard object files or standard shared libraries .
- The main difference is that the libraries aren't automatically loaded at program link time or start-up; instead, there is an API for opening a library, looking up symbols, handling errors, and closing the library.
- C users will need to include the header file <dlfcn.h> to use this API.

DL Library Example

Here's an example from the man page of dlopen(3). This example loads the math library and prints the cosine of 2.0, and it checks for errors at every step :

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

int main(int argc, char **argv) {
    void *handle;
    double (*cosine)(double);
    char *error;

    handle = dlopen ("/lib/libm.so.6", RTLD_LAZY);
    if (!handle) {
        fputs (dlerror(), stderr);
        exit(1);
    }
```

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

int main(int argc, char **argv) {
    void *handle;
    double (*cosine)(double);
    char *error;

    handle = dlopen ("/lib/libm.so.6", RTLD_LAZY);
    if (!handle) {
        fputs (dlerror(), stderr);
        exit(1);
    }

    cosine = dlsym(handle, "cos");
    if ((error = dlerror()) != NULL)  {
        fputs(error, stderr);
        exit(1);
    }

    printf ("%f\n", (*cosine)(2.0));
    dlclose(handle);
}
```

Shared Libraries Can Be Scripts

It's worth noting that the GNU loader permits shared libraries to be text files using a specialized scripting language instead of the usual library format.

This is useful for indirectly combining other libraries. For example, here's the listing of /usr/lib/libc.so :

```
/* GNU ld script
   Use the shared library, but some functions are only in
   the static library, so try that secondarily.  */
GROUP ( /lib/libc.so.6 /usr/lib/libc_nonshared.a )
```

Symbol Versioning and Version Scripts

- Typically references to external functions are bound on an as-needed basis, and are not all bound when the application starts up.
- If a shared library is out of date, a required interface may be missing; when the application tries to use that interface, it may suddenly and unexpectedly fail.

- A solution to this problem are symbol versioning coupled with version scripts.
- With symbol versioning, the user can get a warning when they start their program if the libraries being used with the application are too old.

Removing symbols for space

All the symbols included in generated files are useful for debugging, but take up space. If you need space, you can eliminate some of it.

The best approach is to first generate the object files normally, and do all your debugging and testing first
Afterwards, once you've tested the program thoroughly, use strip(1) to remove the symbols.

Extremely small executables

You might find the paper Whirlwind Tutorial on “Creating Really Teensy ELF Executables for Linux” useful. It describes how to make a truly tiny program executable.
http://www.muppetlabs.com/~breadbox/software/tiny/
http://www.muppetlabs.com/~breadbox/software/tiny/teensy.html

Examples

The following are more examples of all three approaches (static, shared, and dynamically loaded libraries). File libhello.c is a trivial library, with libhello.h as its header. File demo_use.c is a trivial caller of the library.

=>File libhello.c

```
/* libhello.c - demonstrate library use. */

#include <stdio.h>

void hello(void) {
  printf("Hello, library world.\n");
}
```

=>File libhello.h

```
/* libhello.h - demonstrate library use. */


void hello(void);
```

=>File demo_use.c

```
/* demo_use.c -- demonstrate direct use of the "hello" routine */

#include "libhello.h"

int main(void) {
 hello();
 return 0;
}
```

File script_static

```
#!/bin/sh

# Create static library's object file, libhello-static.o.

gcc -Wall -g -c -o libhello-static.o libhello.c

# Create static library.

ar rcs libhello-static.a libhello-static.o

# Compile demo_use program file.

gcc -Wall -g -c demo_use.c -o demo_use.o

gcc -g -o demo_use_static demo_use.o -L. -lhello-static

# Execute the program.
./demo_use_static
```

File script_shared

```
#!/bin/sh
# Shared library demo

# Create shared library's object file, libhello.o.

gcc -fPIC -Wall -g -c libhello.c

# Create shared library.
# Use -lc to link it against C library, since libhello
# depends on the C library.

gcc -g -shared -Wl,-soname,libhello.so.0 \
    -o libhello.so.0.0 libhello.o -lc

# At this point we could just copy libhello.so.0.0 into
# some directory, say /usr/local/lib.
```

```
# Now we need to call ldconfig to fix up the symbolic links.

# Set up the soname.  We could just execute:
# ln -sf libhello.so.0.0 libhello.so.0
# but let's let ldconfig figure it out.

/sbin/ldconfig -n

# Set up the linker name.
# In a more sophisticated setting, we'd need to make
# sure that if there was an existing linker name,
# and if so, check if it should stay or not.

ln -sf libhello.so.0 libhello.so

# Compile demo_use program file.

gcc -Wall -g -c demo_use.c -o demo_use.o
```

```
# Create program demo_use.
# The -L. causes "." to be searched during creation
# of the program; note that this does NOT mean that "."
# will be searched when the program is executed.

gcc -g -o demo_use demo_use.o -L. -lhello

# Execute the program.  Note that we need to tell the program
# where the shared library is, using LD_LIBRARY_PATH.

LD_LIBRARY_PATH="." ./demo_use
```

File demo_dynamic.c

```
/* demo_dynamic.c -- demonstrate dynamic loading and
   use of the "hello" routine */



/* Need dlfcn.h for the routines to
   dynamically load libraries */
#include <dlfcn.h>

#include <stdlib.h>
#include <stdio.h>

/* Note that we don't have to include "libhello.h".
   However, we do need to specify something related;
   we need to specify a type that will hold the value
   we're going to get from dlsym(). */

/* The type "simple_demo_function" describes a function that
   takes no arguments, and returns no value: */

typedef void (*simple_demo_function)(void);
```

```
int main(void) {
const char *error;
void *module;
simple_demo_function demo_function;

/* Load dynamically loaded library */
module = dlopen("libhello.so", RTLD_LAZY);
if (!module) {
  fprintf(stderr, "Couldn't open libhello.so: %s\n",
          dlerror());
  exit(1);
}

/* Get symbol */
dlerror();
demo_function = dlsym(module, "hello");
if ((error = dlerror())) {
  fprintf(stderr, "Couldn't find hello: %s\n", error);
  exit(1);
}
/* Now call the function in the DL library */
(*demo_function)();

/* All done, close things cleanly */
dlclose(module);
return 0;
}
```

File script_dynamic

```
#!/bin/sh
# Dynamically loaded library demo

# Presume that libhello.so and friends have
# been created (see dynamic example).

# Compile demo_dynamic program file into an object file.

gcc -Wall -g -c demo_dynamic.c

# Create program demo_use.
# Note that we don't have to tell it where to search for DL libraries,
# since the only special library this program uses won't be
# loaded until after the program starts up.
# However, we do need the option -ldl to include the library
# that loads the DL libraries.

gcc -g -o demo_dynamic demo_dynamic.o -ldl

# Execute the program.  Note that we need to tell the
# program where get the dynamically loaded library,
# using LD_LIBRARY_PATH.

LD_LIBRARY_PATH="." ./demo_dynamic
```