About This Document

This article writed for Acikkod.ORG. The first version of this document was released on 18 Aug 2004.

Abstract

Libraries provide pre-defined common functions to programmers for modular programming. All modern unix systems accept to use two kind of libraries: Static libraries and shared libraries.

Libraries allow the programmers to writing smaller and modular codes. They allow to update programs easily. Because, if you update a library file, all programs will be updated that are use this library file.

Static Libraries

Static libraries are just object files that are linked into the program during compilation. You don't need to re-compile them to link your program. For that reason, static libraries reduce recompilation time. Nowadays, we have faster computers and recompilation time is not a problem for us. Programmers often use shared libraries because of their advantages. Static libraries usefull for vendors, who don't want to give source code of their libraries. Static libraries end with the “.a” suffix. To create static library, you can use '***ar***' tool.

```
$ ar rc libtest.a test.o util.o
```

This creates a static library that includes *test.o* and *util.o* object files. The 'c' flags means to create library if it does not exist. If there is library file, object files will be added to it. The '**r**' flag tells to ar, replace newer object files with olders.

We should re-index new or modified library files. This process obtain faster control to compiler during compilation. You can use *ranlib(1)* to create index.

```
$ ranlib libtest.a
```

To compile a program with our library, we should use '*-l libraryname*' parameter with *gcc*(1).

```
$ gcc prog.o -L/home/simsek/libtest -ltest
```

Note that, when writing library name after '`-l`' parameter remove 'lib' prefix from library filename. –L parameter tells to gcc directories which search for libraries.

Here is a simple scenario to create and use library:

main.c

```
#include <stdio.h>

/* functions have to be external */
extern void test();
extern void util();

int main() {
  printf("in main");

  /* Let call functions in library */
  test();
  util();

  return 0;
}
```

test.c

```
int test() {
  printf("in test");

  return 0;
}
```

util.c

```
int util() {
  printf("in util");

  return 0;
}
```

Lets compile above source codes:

```
$ gcc -c test.c
$ gcc -c util.c
$ ar rc libtest.a test.o util.o
$ ranlib libtest.a
$ gcc -c main.c
$ gcc main.o -L /home/simsek -ltest -o testprog
```

We compiled source files and produce a executable program named testprog by last command above. If you run testprog, it will produce following outputs:

```
$ ./testprog

in main
in test
in util
```

Shared Libraries

Shared libraries will be inserted to programs when they start to run. Program never includes code from shared libraries. Just includes references to functions in library. All programs that use this library share it.

Shared library has a “*soname*”. So-names are links to real name. Shared library has also a linker name that gcc calls it with this name. Imagine it like a soname without version number.

```
$ ls /usr/lib/libz*
libz.a libz.so.1@ libz.so.1.2.1.1*
```

'*libz.a*' is a static library. *libz.so.1* is a link to *libz.so.1.2.1.1*. libz.so.1 is the soname and libz.so is the linker name.

Programs keep sonames so if you change library real name (because of version) it doesn't affect your programs.

```
$ ldd /bin/ls
        librt.so.1 => /lib/librt.so.1 (0x4001a000)
        libc.so.6 => /lib/libc.so.6 (0x4002c000)
        libpthread.so.0 => /lib/libpthread.so.0 (0x4015b000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

ldd(1), determines library dependence for a program. You can see libraries above that 'ls' directory list program needs. The names are soname.

ldconfig(8) finds out existing libraries and store soname's to */etc/ld.so.cache* file. It checks /lib, /usr/lib and directories specified in /etc/ld.so.conf file. After you install a library, you should type 'ldconfig' command and regenerate cache file.

To create shared library, you have to use `-fPIC` or `-fpic` options beside `-shared` options. A function in shared library will be placed diffirent address in memory for each program. For that reason we have to generate position indipendent code (PIC). `-fPIC` option generates larger object file then `-fpic`. `-fpic` generates smaller and faster objects. But they will be platform dependent and they will include less debug informations.

```
$ gcc -c -fpic test.c
$ gcc -shared -o libtest.so test.o
$ ls -l libtest.so
-rwxr-xr-x 1 simsek users 6077 2004-08-17 23:04 libtest.so*
$ ldconfig
```

I prepared a shared library from our test.c. Now we can use test() function in our other programs that compiled with libtest. You can see an example program that uses shared function test().

shared.c

```
#include <stdio.h>

/* functions have to be external */
extern void test();

int main() {

  printf("in main");

  /* Let call function from shared library */
  test();

  return 0;

}
```

```
# gcc -c shared.c
# gcc -o sharedprog shared.o -L. -ltest
# ./sharedprog

in main
in test
```

test() function in shared library produced '*in test*' message.

The linker will search libraries under /lib and /usr/lib compilation duration. If our library is in another directory, it will not find it. We'll solve this problem with gcc. If you give '`-Wl,option`' parameter to gcc, gcc will pass 'option' directly to the linker. We'll use this feature to pass library directory to linker with '`-rpath`' option.

```
# gcc -o sharedprog shared.o -L. -ltest -Wl,-rpath,/home/simsek/libtest
```

Another solution of above problem is using `LD_LIBRARY_PATH` shell variable. If you add directory that contains your libraries to this environment variable, linker will use it.

```
# export LD_LIBRARY_PATH=/lib:/usr/lib:/home/simsek/libtest
```

Baris Simsek
EnderUNIX Software Development Team
http://www.enderunix.org/simsek/