ACRC How-To: Linking to Libraries (on Linux)

Introduction

The process of creating an executable program has three steps:

- **Preprocessing**: Source code manipulation by file & conditional inclusion and macro substitution.
- **Compilation**: Transformation of (human readable) source code into (binary) object code.
- **Linkage**: Combination of one or more files of object code to create an executable program.

For convenience, several object files can be collected together, prior to linkage, to form libraries. This offers the programmer several advantages. Notably:

- Libraries need only be compiled once and thereafter may be used many times. This can significantly shorten the compilation time for any executable using the library.
- A programmer can make use of code which someone else has invested time in writing, debugging and optimising. Given an API (Application Programmer Interface) we can use libraries 3rd party libraries with little or no knowledge of the details of their implementation.

There are two forms of linkage:

- **Static**: All symbols are resolved at link-time. The effect of this is that all objects are copied and combined to form a stand-alone executable.
- **Dynamic**: Some symbols are left unresolved until run-time. In practice, this means that the executable must locate and complete linkage to any objects when it is run.

There are pros and cons for both approaches. Statically linked executables are large, but you can be certain that all libraries are present and of the correct version. Dynamically linked executables are smaller and can be updated on-the-fly (to maintain operating system security, for example), but their dependencies are not *guaranteed* to be satisfied at run-time.

Libraries suitable for these two different forms of linkage are prepared differently. For dynamic linkage the libraries are called *shared-objects* and are given the file extension `.so`. For static linkage the files are called *archives* and are given the `.a` file extension.

Examples

In this section I will demonstrate how to create an executable which calls a routine from a library. I will show examples—in both Fortran and C—of calling LAPACK's dgesv routine (to solve a linear system of equations via the method of LU decomposition). The examples will use GNU compilers. By default, dynamic linkage will be preferred by the compiler and so I will demonstrate that process first.

Dynamic Linkage using OpenBLAS

Let's suppose that I have a version of OpenBLAS (http://xianyi.github.com/OpenBLAS) installed under my home directory and two sample programs; dgesv.f90 and dgesv.c. (See the appendix for more details of the source code and how to compile and install the library.) I can compile and link either of these programs to the library as shown below.

The OpenBLAS build creates both shared-objects and archives:

```
% ls $HOME/openblas/0.2.5/lib
libopenblas.a  libopenblas_nehalemp-r0.2.5.a  libopenblas_nehalemp-
r0.2.5.so  libopenblas.so  libopenblas.so.0
```

Fortran

```
gfortran dgesv.f90 -L$HOME/openblas/0.2.5/lib -lopenblas \
-o dgesv-fort-openblas.exe
```

C

```
gcc dgesv.c -I$HOME/openblas/0.2.5/include \
-L$HOME/openblas/0.2.5/lib -lopenblas -o dgesv-c-openblas.exe
```

Modern compilers can perform both the compile and link steps and the above commands show both these steps done in a single invocation of the respective compilers.

Notice that paths to directories containing library files are given with the `-L` option and the names of particular libraries to link to are given with the `-l` option (minus the `lib` prefix and any file extension).

Common Problems

As stated above, dynamic linkage will be preferred by the compiler. At this point, the executable cannot be run if the requisite shared-objects cannot be located at run time. For example:

```
% ./dgesv-fort-openblas.exe
./dgesv-fort-openblas.exe: error while loading shared libraries:
libopenblas.so.0: cannot open shared object file: No such file or
directory
```

We can check whether the dependencies of a dynamically linked executable can be located using the `ldd` command:

```
% ldd dgesv-fort-openblas.exe
      linux-vdso.so.1 =>  (0x00007ffff97ff000)
      libopenblas.so.0 => not found
      libgfortran.so.3 => /usr/lib64/libgfortran.so.3
(0x0000003f8e400000)
      libm.so.6 => /lib64/libm.so.6 (0x00000033c9800000)
      libgcc_s.so.1 => /lib64/libgcc_s.so.1 (0x0000003f8f200000)
      libc.so.6 => /lib64/libc.so.6 (0x00000033c9400000)
      /lib64/ld-linux-x86-64.so.2 (0x00000033c9000000)
```

Corroborating the run-time error, we see that `libopenblas.so.0` cannot be found, in this case.

This situation can be addressed in one of two ways. The first is to augment your `LD_LIBRARY_PATH` environment variable. We can do this in the BASH shell using:

```
% export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$HOME/openblas/0.2.5/lib
```

(Where it is best to place this command in your `~/.bashrc` file, if you intend on using the library regularly).

Gethin Williams 2013

Another approach is to tell the executable where it can find the shared-object come run-time. In this case, we augment our compile-and-link command with an option of the form `-Wl,-rpath,/path/to/lib`:

```
%gcc dgesv.c -I$HOME/openblas/0.2.5/include -L$HOME/openblas/0.2.5/lib \
-Wl,-rpath,$HOME/openblas/0.2.5/lib -lopenblas -o dgesv-c-openblas.exe
```

In both cases, if we re-run the `ldd` command, we will see that all the dependencies have been resolved:

```
% ldd dgesv-c-openblas.exe
      linux-vdso.so.1 =>  (0x00007fff199ff000)
      libopenblas.so.0 =>
/home/ggdagw/openblas/0.2.5/lib/libopenblas.so.0 (0x00002ae93bbf1000)
      libc.so.6 => /lib64/libc.so.6 (0x00000033c9400000)
      libm.so.6 => /lib64/libm.so.6 (0x00000033c9800000)
      libpthread.so.0 => /lib64/libpthread.so.0 (0x00000033ca000000)
      libgfortran.so.3 => /usr/lib64/libgfortran.so.3
(0x0000003f8e400000)
      /lib64/ld-linux-x86-64.so.2 (0x00000033c9000000)
```

and the executable will run successfully:

```
% ./dgesv-c-openblas.exe
-15.000000
8.000000
2.000000
```

Static linkage using Netlib's LAPACK

For this example, I'll use the reference distribution of LAPACK from Netlib (http://www.netlib.org/lapack), as it builds only archive files by default.

Fortran

```
gfortran dgesv.f90 -L$HOME/lapack/lapack-3.4.2 -llapack -lrefblas \
-o dgesv-fort-netlib.exe
```

C

```
gcc dgesv.c -I$HOME/lapack/lapack-3.4.2/lapacke/include \
-L$HOME/lapack/lapack-3.4.2 -llapacke -llapack -lrefblas -lgfortran \
-o dgesv-c-netlib.exe
```

`liblapacke.a` contains the LAPACK C interface. Notice that we must also link our executable the `gfortran` library (as it contains routines injected by the compiler into the other archive files).

We can verify that static linkage has occurred by comparing file sizes. Statically-linked executable files are larger than their dynamically-linked counterparts:

```
% ls -lh dgesv-fort-*.exe
-rwxr-xr-x 1 ggdagw isys  29K Mar 25 16:47 dgesv-fort-netlib.exe
-rwxr-xr-x 1 ggdagw isys 9.4K Mar 22 15:41 dgesv-fort-openblas.exe
```

Common Problems

The successful linkage above relies upon the given order of the libraries. In the abstract case, if a library A depends upon symbols defined in library B, then A must appear first in the list supplied to the linker. In our case, LAPACK routines call BLAS routines and so `liblapack.a` must be appear before `librefblas.a` in the given list. If we disrupt that ordering we will precipitate an error:

```
% gfortran dgesv.f90 -L$HOME/lapack/lapack-3.4.2 -lrefblas -llapack -o
dgesv-fort-netlib.exe
/home/ggdagw/lapack/lapack-3.4.2/liblapack.a(dgetrf.o): In function
`dgetrf_':
dgetrf.f:(.text+0x3f3): undefined reference to `dtrsm_'
dgetrf.f:(.text+0x4d0): undefined reference to `dgemm_'
/home/ggdagw/lapack/lapack-3.4.2/liblapack.a(dgetrs.o): In function
`dgetrs_':
dgetrs.f:(.text+0x18b): undefined reference to `dtrsm_'
dgetrs.f:(.text+0x1fd): undefined reference to `dtrsm_'
dgetrs.f:(.text+0x375): undefined reference to `dtrsm_'
dgetrs.f:(.text+0x3e7): undefined reference to `dtrsm_'
/home/ggdagw/lapack/lapack-3.4.2/liblapack.a(dgetf2.o): In function
`dgetf2_':
dgetf2.f:(.text+0x1cb): undefined reference to `idamax_'
dgetf2.f:(.text+0x21d): undefined reference to `dswap_'
dgetf2.f:(.text+0x38c): undefined reference to `dger_'
dgetf2.f:(.text+0x3f0): undefined reference to `dscal_'
collect2: ld returned 1 exit status
```

Another problem commonly encountered is mismatched symbols. When a source code file is compiled and turned into object code, the names of routines are 'mangled'. They are typically decorated with leading or trailing underscore (which we can see in the last example). Unfortunately different compilers often use different mangling strategies and so libraries compiled with compiler A are not guaranteed to link with other objects compiled with compiler B. In general it is highly advisable to ensure that all objects given to the linker have been prepared using the same compiler.

If you receive '`undefined reference`' errors, and have passed the linker all the required libraries, then different name-mangling is a likely source of the problem. We can simulate the use of different compiler by instructing `gcc` to adopt a different name-mangling strategy for the program code compared to that used when creating the library:

```
% gcc dgesv.c -fleading-underscore -I$HOME/lapack/lapack-
3.4.2/lapacke/include -L$HOME/lapack/lapack-3.4.2 -llapacke -llapack
-lrefblas -lgfortran -o dgesv-c-netlib.exe
/usr/lib/gcc/x86_64-redhat-linux/4.4.6/../../../../lib64/crt1.o: In
function `_start':
(.text+0x20): undefined reference to `main'
/tmp/ccaognV1.o: In function `_main':
dgesv.c:(.text+0x11a): undefined reference to `_LAPACKE_dgesv'
dgesv.c:(.text+0x146): undefined reference to `_printf'
```

Forcing Static Linkage

If a library build contains both shared-objects and archives, we can force static linkage with the `-static` flag.

Fortran

```
% gfortran dgesv.f90 -L$HOME/openblas/0.2.5/lib -lopenblas -lpthread \
-lgfortran -static -o dgesv-fort-openblas-static.exe
```

C

```
% gcc dgesv.c -I$HOME/openblas/0.2.5/include \
-L$HOME/openblas/0.2.5/lib -lopenblas -lpthread -lm -static \
-o dgesv-c-openblas-static.exe
```

Notice, that we have had to augment the list of libraries passed to the linker (`-lpthread -lgfortran`). If the corresponding archives are not found on your system, you will receive an error. Upon successful linkage, when we examine the executable using `ldd` we see:

```
% ldd dgesv-fort-openblas-static.exe
      not a dynamic executable
```

Appendix

Example code

dgesv.f90:

```
Program DirectSolve

  ! LAPACK
  ! Perform a direct solve for the equation A*x=b, using LU decomposition.
  ! A is a general, double precision matrix.
  implicit none

  ! declarations, NB double precision
  integer, parameter      :: N = 3
  integer, parameter      :: LDA = N   ! leading dimension of A
  integer, parameter      :: LDB = N   ! leading dimension of B
  integer, parameter      :: NRHS = 1 ! no. of RHS, i.e columns in b
  integer, dimension(N) :: IPIV
  integer                 :: INFO
  integer                 :: ii
  logical, parameter    :: verbose = .false.
  real(kind=8), dimension(LDA,N)       :: A ! LDAxN matrix
  real(kind=8), dimension(LDB,NRHS) :: B ! LDBxNRHS matrix

  ! insert values into matrix A:
  ! ( 1  3 -2)
  ! ( 3  5  6)
  ! ( 2  4  3)
  A(1,1) =  1.0d+0
  A(1,2) =  3.0d+0
  A(1,3) = -2.0d+0
  A(2,1) =  3.0d+0
  A(2,2) =  5.0d+0
  A(2,3) =  6.0d+0
  A(3,1) =  2.0d+0
  A(3,2) =  4.0d+0
  A(3,3) =  3.0d+0
```

```
  ! insert values into matrix B:
  ! ( 5)
  ! ( 7)
  ! ( 8)
  B(1,1) =  5.0d+0
  B(2,1) =  7.0d+0
  B(3,1) =  8.0d+0

  ! solve (using LU decomposition) using LAPACK's DGESV routine.
  ! NB the known vectors B will be exchanged, in place, with the
  ! solution vectors X, on exit.
  call dgesv(N, NRHS, A, LDA, IPIV, B, LDB, INFO)

  ! check the value of info
  if (INFO .ne. 0) then
     write(*,*) 'ERROR calling DGETRF'
     stop
  endif

  ! Print the result vector X.
  ! It should be:
  ! (-15)
  ! ( 8)
  ! ( 2)
  do ii=1,LDB
     write(*,*) B(ii,1)
  end do

if (verbose) then
     ! Also print out A, the encoding of the LU decomposition:
     write (*,*) ''
     write (*,*) 'LU decomposition as encoded in matrix A:'
     do ii=1,N
        write(*,*) A(ii,:)  ! all cols for a given row
     end do

     write (*,*) ''
     write (*,*) '..and the IPIV vector:'
     write (*,*) IPIV
  end if

end Program DirectSolve
```

degsv.c:

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

int main (int argc, const char * argv[])
{
  double a[3*3] = {1,3,2,3,5,4,-2,6,3};
  double b[3*1] = {5,7,8};
  lapack_int info,n,lda,ldb,nrhs,ipiv[3];
  int ii;

  n    = 3;
  nrhs = 1;
  lda  = 3;
  ldb  = 3;

  info = LAPACKE_dgesv(LAPACK_COL_MAJOR,n,nrhs,a,lda,ipiv,b,ldb);

  for(ii=0; ii<n; ii++)
   {
    printf("%lf\n",b[ii]);
   }
  return(info);
}
```

How to build the libraries

OpenBLAS:

```
% cd $HOME
% mkdir openblas
% cd openblas
% wget http://github.com/xianyi/OpenBLAS/tarball/v0.2.5 \
-O openblas.tar.gz
% tar -xzf
% cd xianyi-OpenBLAS-93dd133
% make FC=gfortran
% make PREFIX=$HOME/openblas/0.2.5 install
```

Netlib's LAPACK:

```
% cd $HOME
% mkdir lapack
% cd lapack
% wget http://www.netlib.org/lapack/lapack-3.4.2.tgz
% tar -xzf lapack-3.4.2.tgz
% cd lapack-3.4.2
% cp make.inc.example make.inc
% make lapacklib blaslib
% cd lapacke; make
```

SLATEC:

Not referenced in the above examples, I will additionally describe how to build the SLATEC (http://www.netlib.org/slatec/) library:

```
% cd $HOME
% mkdir slatec
% wget http://www.netlib.org/slatec/slatec_src.tgz
% tar -xzf slatec_src.tgz
% cd src
% wget http://www.netlib.org/slatec/slatec4linux.tgz
% tar -xzf slatec4linux.tgz
% export FC=gfortran
% make
```

Following this procedure you will have built the following archive and shared object:

- `$HOME/slatec/src/static/libstatic.a`
- `$HOME/slatec/src/dynamic/libstatic.so`