Compiling and Linking

Compiling and Linking

- Daniel Trahan
- *Email: Daniel.Trahan@Colorado.edu*
- *RC Homepage: https://www.colorado.edu/rc*

Sign in! http://tinyurl.com/curc-names

- Slides available for download at: https://github.com/ResearchComputing/RMACC-2019_Compiling_and_Linking

Adapted from slides by Nick Featherstone

Outline

- Compiling vs. Linking
- Shared vs. Static Libraries
- Creating static and shared libraries
- Environmental Considerations
  - LD_LIBRARY_PATH
  - ldd and nm
  - RPATH
- Link order

Before we begin

- `ssh username@tlogin1.rc.colorado.edu`
- `ssh scompile`
- `ml intel`
- `cd /projects/$USER`
- `git clone https://github.com/ResearchComputing/RMACC-2019_Compiling_and_Linking.git` (all one line)
- `cd RMACC-2019_Compiling_and_Linking`

Nano survival guide

- Ctrl+o save (need to confirm filename)
- Ctrl+x exit
- Ctrl+k cut
- Ctrl+u paste

Compiling vs. Linking

- Compilation generates object files (machine code) from source code
- Linking bundles object files together to generate a complete executable
- Often carried out via single command for small projects.
- Multiple stages for complex projects

Compilation/Linking Examples

Single-Step

```
icc -I . hello.c hellofunc.c -o hello.exe
```

Separate Steps

```
icc -I . -c hellofunc.c     -o hellofunc.o
icc -I . -c hello.c         -o hello.o

icc  hello.o hellofunc.o -o hello.exe
```

compilation

linking

Whats Behind the Curtain?

- Quite a bit going on in the background
- Quick exercise:
  - In `sample_make` directory, edit `machine.def`
    - Change `-O2` to `-O2 -v`
  - Run: `make clean`
  - Run: `make`
- “-v” denotes “verbose”

Compiler Optimization

- Compilers usually have some built in automatic optimization flags that can be passed at compile time.
  - `-O0` – No optimization applied. Useful if Optimization creates inaccuracies in code.
  - `-O1` – Small optimizations: Data flow analysis, code motion, and instruction scheduling
  - `-O2` – Heavy optimizations: loop unrolling, tail recursion, variable renaming
  - `-O3` – Aggressive loop optimizations. Useful on apps with heavy floating point evaluating loops.
- Architecture specific instruction sets:
  - `-xCORE-AVX` – Haswell processors
  - `-xCORE-AVX512` – Xeon phi and Skylake processors

Libraries

- Common practice to bundle related functionality into single library
- Can be used in multiple programs
- Two flavors
  - Shared (dynamically linked at runtime)
  - Static (bundled into executable at compile time)

Shared Libraries

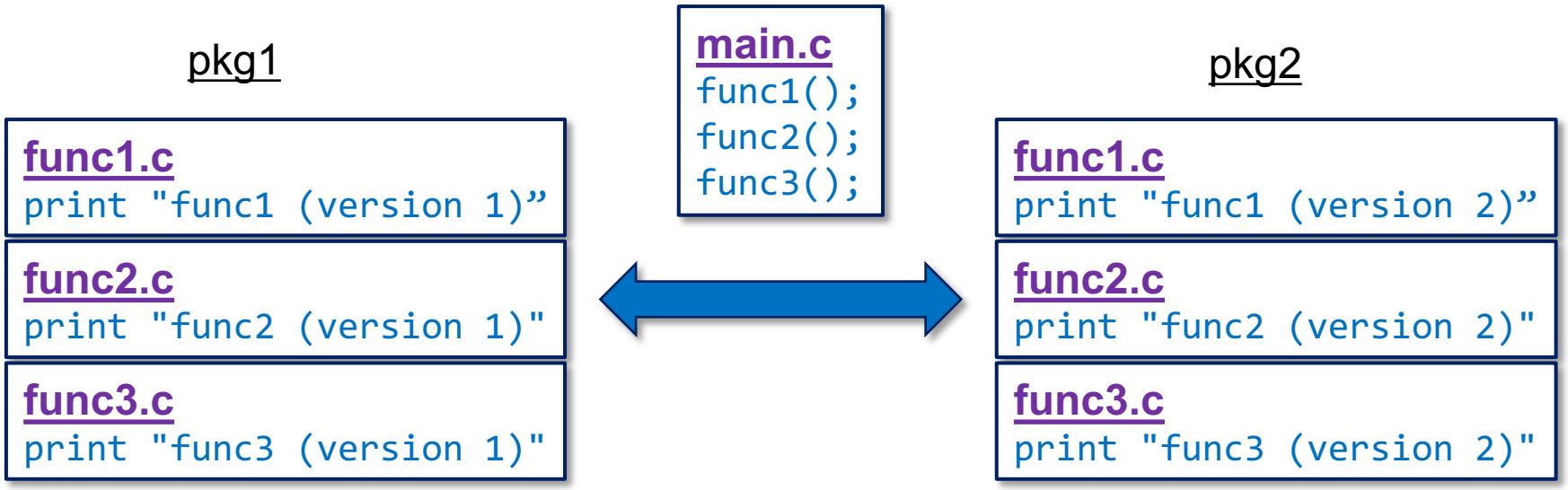
- .so extension (e.g., `libc.so`)
- Advantages
  - Loading controlled via environment
  - HPC centers often provide optimized versions of commonly-used libraries.
  - Smaller executable size
  - RAM efficient (multiple processes can access same shared library)
- Disadvantages:
  - occasional version conflicts
  - User must be “environment-aware”
- Advice:
  - Use shared libraries when incorporating 3rd party software into your project.

Static Libraries

- .a extension (e.g., liblapack.a)
- Advantages
  - More difficult to introduce version conflicts
  - Conflicts discovered at compiled time
- Disadvantages:
  - Larger executable size
  - Potential for redundant code in memory
  - More difficult to incorporate updates
- Advice:
  - Where natural, us static libraries to bundle portions of your own project.
  - Use static libraries for 3rd party packages that tend to change significantly between versions (e.g., Boost)

Hands-On Exercise

- We are going to work with a small project that uses statically- and dynamically-linked libraries which we create.
- We will examine how to switch between shared libraries in pkg1 and pkg2 at run-time



Our pkg1 Makefile

- Change to the directory “libraries” this is our project directory.
- run: `mkdir pkg1/lib`
- run: `mkdir pkg1/include`
- Open `pkg1/src/Makefile`
- Let’s have a look.
- Presently, instructions for creating the shared and static library are missing…

Creating a Static Library

- Step1: Compile object files needed by library

```
icc -I . -O1 -c func1.c    -o func1.o
icc -I . -O1 -c func2.c    -o func2.o
icc -I . -O1 -c func3.c    -o func3.o
```

type this at command line

- Step2: Bundle your .o files using the archiver

```
ar rc libstatic.a func1.o func2.o func3.o
```

- GNU archiver options:
  - r – replace functions if already found to exist in library
  - c – create static library if it does not already exist

On Your Own

- Modify the (now empty) rule for `$(LIBSTATIC)` in your Makefile so that it:
  - Compiles the func.o files
  - Creates an archive from the func.o files
  - Use the variables defined in the Makefile and in machine.def
- Test it:
  - `make libstatic.a`

Creating a Shared Library

- Step1: Compile object files with -fPIC

```
icc -I . -O1 -fPIC -c func1.c    -o func1.o
icc -I . -O1 -fPIC -c func2.c    -o func2.o
icc -I . -O1 -fPIC -c func3.c    -o func3.o
```

type this at command line

- Step2: Use icc with –shared flag to create .so file

```
icc -shared func1.o func2.o func3.o -o libshared.so
```

- Shared library options:
  - -fPIC – generate Position Independent Code
  - -shared – create a shared library

On Your Own

- Modify the (now empty) rule for `$(LIBSHARED)` in your Makefile so that it:
  - Compiles the func.o files
  - Creates a shared library from the `func.o` files
  - Use the variables defined in the Makefile and in machine.def
- Test it:
  - `make libshared.so`
  - `make clean`
  - `make all` (will build both libs and “install”)

Creating package 2

- Change back to the “libraries” directory
- `cp -r pkg1 pkg2` (create pkg2 directory)
- Change to `pkg2/src`
- Modify each funcX.c to say “version 2”
- `make clean` then `make all`
- Change back to the “libraries” directory

Linking our Libraries

- Static and shared libraries are linked with different syntax at compile time
- Static libraries – provide full path to library:

```
icc -O2 -I . -I pkg1/include main.c pkg1/lib/libstatic.a -o test.static
```

- Shared libraries:
  - Indicate directory with –L flag
  - Indicate library using “l” shorthand (libshared.so = -lshared)

```
icc -O2 -I . -I pkg1/include main.c -Lpkg1/lib -lshared -o test.dynamic
```

- Try this at the command line

On Your Own

- Let's examine the Makefile in "libraries"
- Edit the `LIBSTATIC` and `LIBSHARED` variables so that test.dynamic and test.static build correctly.
- Run make clean & make to build
- Try it out:
  - ./test.static
  - ./test.dynamic
- What happens?

LD_LIBRARY_PATH

- Test.dynamic failed because the system did not know where to look for libshared

- Using dynamic libraries is a two-step process
  - Compile time: compiler pointed to library via –L
  - Run time: system checks system defaults and LD_LIBRARY_PATH directories for shared libraries

- LD_LIBRARY_PATH
  - Colon-separated list of directories
  - System checks first directory, then second, and so on when loading shared libraries at runtime.

LD_LIBRARY_PATH

- Example:
  - LD_LIBRARY_PATH = /lib:/home/mylib
  - System checks
    1. /lib first
    2. /home/mylib second
- Example:
  - LD_LIBRARY_PATH = /home/mylib:/usr/lib64
  - System checks
    1. /home/mylib first
    2. /lib second

Running Test.dynamic

- Save original LD_LIBRARY_PATH:

```
export LD_SAVE=$LD_LIBRARY_PATH
```

- Try this:

```
export LD_LIBRARY_PATH=pkg1/lib:$LD_SAVE
./test.dynamic
```

- And this:

```
export LD_LIBRARY_PATH=pkg2/lib:$LD_SAVE
./test.dynamic
```

Readelf: What is Needed?

- ELF = Executable Linkable Format
- Standard format for Linux executables and libraries
- Readelf: utility for parsing ELF files
- Tells us which dynamic libraries are needed
- `readelf -h <filename>` ***view header***
- `readelf -d <filename>` ***view dynamic section***
- Try it with test.dynamic and test.static

Readelf Output

readelf –d test.static

```
Dynamic section at offset 0x35c0 contains 27 entries:
  Tag        Type                         Name/Value
 0x0000000000000001 (NEEDED)             Shared library: [libm.so.6]
 0x0000000000000001 (NEEDED)             Shared library: [libgcc_s.so.1]
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
 0x0000000000000001 (NEEDED)             Shared library: [libdl.so.2]
```

readelf –d test.dynamic

```
Dynamic section at offset 0x35c0 contains 28 entries:
  Tag        Type                         Name/Value
 0x0000000000000001 (NEEDED)             Shared library: [libshared.so]
 0x0000000000000001 (NEEDED)             Shared library: [libm.so.6]
 0x0000000000000001 (NEEDED)             Shared library: [libgcc_s.so.1]
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
 0x0000000000000001 (NEEDED)             Shared library: [libdl.so.2]
```

LDD: Where am I Pointed?

- The ldd utility indicates which shared library will be used
- Depends on system defaults and current LD_LIBRARY_PATH
- Try it out:
  - `ldd test.static`
  - `ldd test.dynamic`

ldd Output

ldd test.static

```
linux-vdso.so.1 =>  (0x00007fff584c1000)
libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007fe0d3e40000)
libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007fe0d3c2a000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fe0d3860000)
libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007fe0d365c000)
/lib64/ld-linux-x86-64.so.2 (0x00007fe0d4149000)
```

ldd test.dynamic

```
linux-vdso.so.1 =>  (0x00007ffd11ddd000)
libshared.so => pkg1/lib/libshared.so (0x00007f379221b000)
libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007f3791f12000)
libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007f3791cfc000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f3791932000)
libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f379172e000)
… (various Intel libraries here)
/lib64/ld-linux-x86-64.so.2 (0x00007f379241c000)
```

Exercise

- Sometimes dynamic linking creates unexpected problems.
- Copy pkg1 to a new directory pkg3
- Modify the FUNCS variable to omit func3.o
- Rebuild the library via: make clean / make
- Point LD_LIBRARY_PATH at pkg3/lib and re-run test.dynamic

Checking Symbols with nm

- Can use the nm utility to check symbols contained within a file
- e.g., nm pkg1/lib/libshared.so
- Examine the pkg1 and pkg3 libraries
- pkg1 shows the func3 symbol
- pkg3 does not

Checking Symbols with nm

- Examine test.dynamic and test.static
- What's different w/ respect to func1,2,3?
- 'T' indicates symbol defined
- 'U' indicates symbol undefined (program must look elsewhere)

RPATH: Hard-Coded Dynamic Linking

- If desired, we can both:
  - link dynamically
  - ignore LD_LIBRARY_PATH
- To do so, specify the RPATH when compiling, e.g.

```
-Wl,-rpath,$(PWD)/pkg1/lib
```

- RPATH supersedes LD_LIBRARY_PATH
- Examine libraries/Makefile

Running test.rpath

- Save original LD_LIBRARY_PATH:

```
export LD_SAVE=$LD_LIBRARY_PATH
```

- Try this:

```
export LD_LIBRARY_PATH=pkg1/lib:$LD_SAVE
./test.rpath
```

- And this:

```
export LD_LIBRARY_PATH=pkg2/lib:$LD_SAVE
./test.rpath
```

The ORIGIN Variable

- The ORIGIN variable allows us to specify a relative RPATH
- This makes it easier to copy code and shared libraries around.
- Compiler flags (double $ is correct):

```
-Wl,-rpath,"\$$ORIGIN"/lib
```

- Run: readelf –d test.origin

ORIGIN: Try it Out

- We can now move our executable and library wherever we like
- test.origin will look for .so file in lib subdirectory

```
mkdir $USER/origin

mkdir $USER/origin/lib

cp test.origin $USER/origin/.

cp pkg1/lib/libshared $USER/origin/lib

/$USER/origin/test.origin
```

Link Order

- Can link multiple libraries when compiling a program. E.g.,

```
ifort main.f90 -o prog1 -lv1 -lv2 -lv3
```

- If libraries contain unique subroutines, order does not matter.

- If two or more libraries contain same subroutine names, it matters.

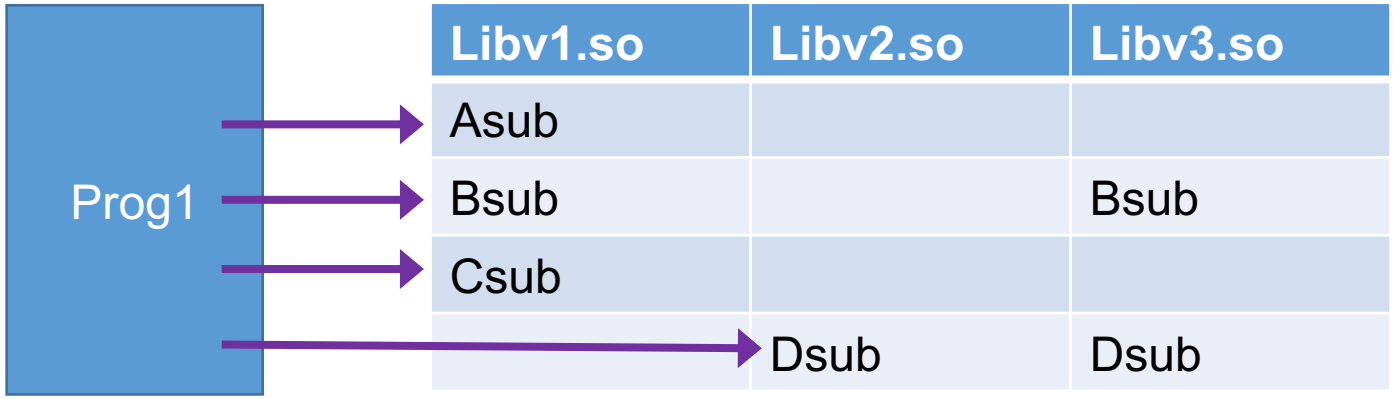
- Why would you have the same routine name?
  - Example: LAPACK & IBM ESSL
    - ESSL & LAPACK both have DGEMM
    - Only LAPACK has dgetrf (band solve routine)
    - May want to use ESSL DGEMM along with LAPACK dgetrf

Link Order

- Let's examine the Makefile in the link_order directory
- Build the code: make clean / make
- Export LD_LIBRARY_PATH=lib:$LD_LIBRARY_PATH
- Run prog1: ./prog1

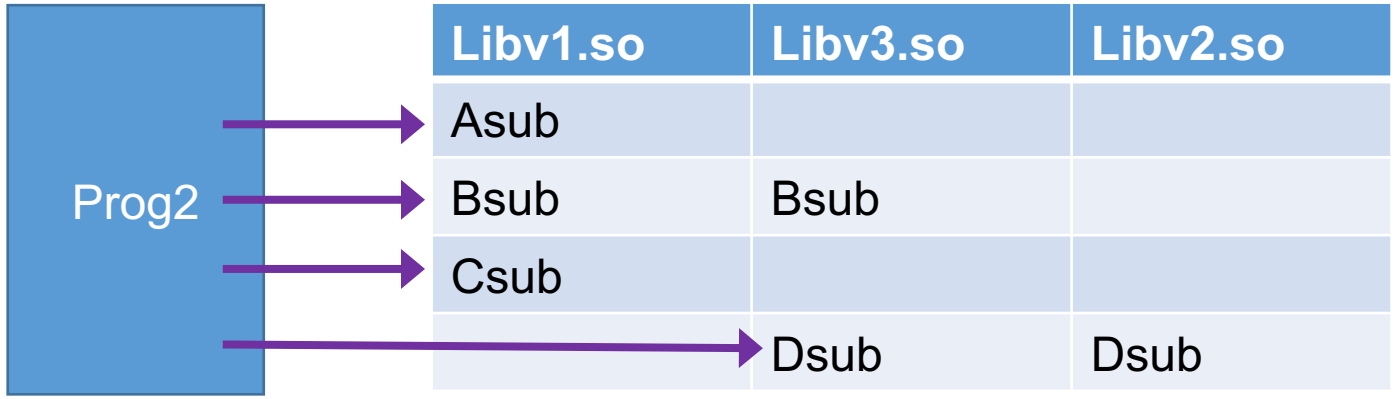
Link Order

- How does order of linking matter?
- Most linkers check from left to right
- Stops once symbol is found
- Good practice: put “root” dependency last
- Ex: `ifort main.f90 -o prog1 -lv1 -lv2 -lv3`



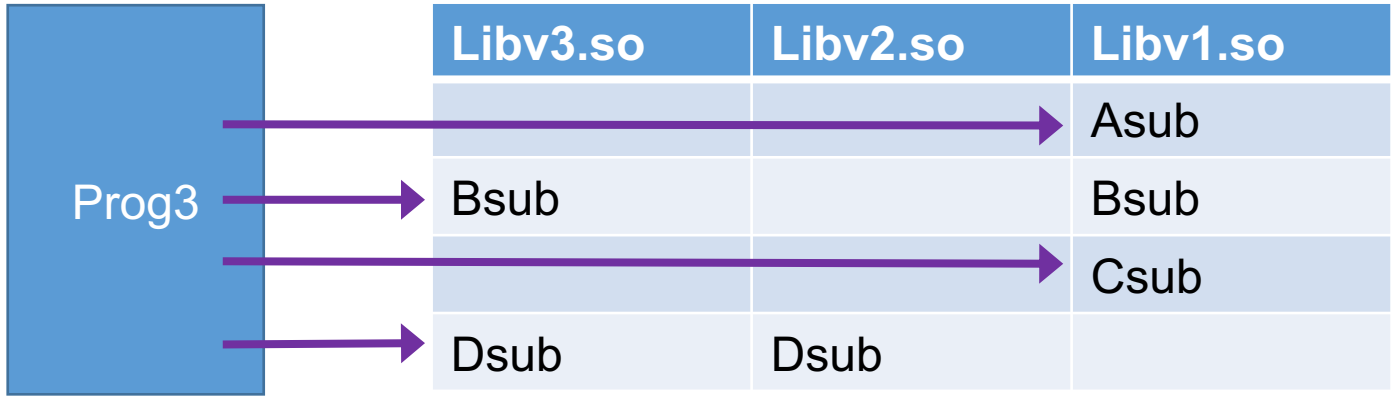
Link Order

- Run prog2…
  - `ifort main.f90 -o prog2 -lv1 -lv3 -lv2`



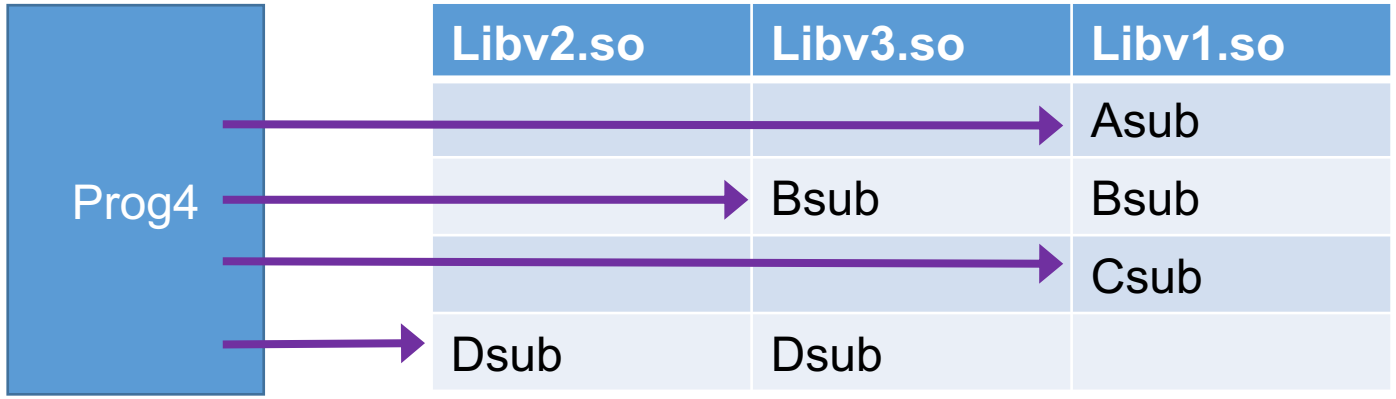
Link Order

- Run prog3…
  - `ifort main.f90 -o prog2 -lv1 -lv3 -lv2`



Link Order

- Run prog4…
  - `ifort main.f90 -o prog2 -lv2 -lv3 -lv1`



Test Yourself

- What will be the output of
  - `ifort main.f90 -o prog1 -lv2 -lv1 -lv3`
  - `ifort main.f90 -o prog1 -lv3 -lv1 -lv2`

Libv1.so	Libv2.so	Libv3.so
Asub		
Bsub		Bsub
Csub		
	Dsub	Dsub

Thank You!

- Survey: http://tinyurl.com/curc-survey18
- Slides: https://github.com/ResearchComputing/RMACC-2019_Compiling_and_Linking