Building programs and libraries

Martin Čuma and Zhiyu (Drew) Li
CHPC User Services

Overview

- Creating an executable
- Building packages
- Program libraries
- Static vs. Dynamic libraries
- Create static & dynamic libraries
- Link with the above libraries to generate an executable.

Getting the exercise files

- For today's exercises, open a session to one of the cluster interactives and run the following commands:

```
cp ~u0101881/talks/LinuxBuilding.tar.gz .
tar  -zxvf  LinuxBuilding.tar.gz
cd LinuxBuilding/
```

Interpreted vs. Compiled code

- Source := collection of ***human-readable*** computer instructions written in a programming language
  (e.g. C, C++, Fortran, Python, R, Java,…)
- Executable := ***binary*** program that can be directly executed on a computer

- **Interpreted** languages: the interpreter parses the source code & executes it immediately
- **Compiled** languages: the source code needs to be transformed into an executable through a chain of compilation & linking
- A few examples:
  a. interpreted languages: Python, R, Matlab, Bash, Tcsh,…
  b. compiled languages: C, C++, Fortran, …

Creating an executable

- Either in 1 step:
  a. `gcc -o simple simple.c`
- Or in 2 steps:
  a. `gcc -c simple.c`
  b. `gcc -o simple simple.o`

or more **generally (C, C++, Fortan)**:

- 1-step:
  a. `$COMPILER -o $EXE  $SOURCE_FILES.{f90,c,cpp}`
- 2-step:
  a. `$COMPILER -c $SOURCE.{f90,c,cpp}`
  b. `$COMPILER -o $EXE $SOURCE.o`

Compilers

- Compilers are system-specific, but, there are quite a few vendors (CHPC has three):
- GNU: `gcc`, `g++`, `gfortran` – open source, free
- Intel: `icc`, `icpc`, `ifort` – commercial, free
- Nvidia (PGI): `nvcc`, `nvc++`, `nvfortran` – commercial, free

Creating an executable (details)

- For compiled languages, the creation of an executable goes through the following steps:
  - *Preprocessing*: the pre-processor takes the source code (`.c`, `.cc`, `.f90`) and "deals" with special statements e.g. `#define`, `#ifdef`, `#include` (C/C++)
  - *Compilation*: takes the pre-processor output and transforms it into assembly language (`*.s`)
  - *Assembly*: converts the assembly code (`*.s`) into machine code/object code (`*.o`)
  - *Linking*: the linker takes the object files (`*.o`) and transforms them into a library (`*.a`, `*.so`) or an executable

- Example : `simple.c` (C source file)
- Pre-processing:

  - `cpp simple.c –o simple.i` **or**
  - `gcc –E simple.c  –o simple.i`

- Compilation: creating assembly code

  - `gcc –S simple.i [ –o simple.s]`

  # can also use `gcc –S simple.c [–o simple.s]`

- Assembly phase: creation of the machine code

  - **`gcc –c simple.c [–o simple.o]`** **or**
  - `as simple.s –o simple.o`

  # can also use `gcc –c simple.s [–o simple.o]`

- Linking: creation of the executable

  - **`gcc simple.c [–o simple]`** **or**
  - **`gcc simple.o [–o simple]`** **or**

  # use ld (the linker as such) -> complicated expression

Optimization and debugging

- The compiler can perform optimizations that improve performance.
  - common flags `-O3` (GNU), `-fast` (Intel), `-fastsse` (Nvidia)
  - Beware! -`O3`,etc can sometimes cause problems (numerical inaccuracies due to optimizations)
- In order to debug program in debugger, symbolic information must be included
  - flag `-g`
  - The easiest debugging is to just add `printf` or `write` statements (like using echo)

Exercise 1

Go to the subdirectory "ex1". There are a few source files in this directory. Compile these programs using the following steps:

1. Compile cpi_ser.c using gcc. Perform the compilation first in **2** steps i.e. create first an object file & then an executable. Perform the same compilation in **1** step.
2. Try the same for pi3_ser.f (Fortran – gfortran). Does it work?
3. Create the object file of ctimer.c with gcc. Then link both object file ctimer.o and pi3_ser.o into an executable using gfortran.
4. Try compiling cpi_ser.c with the optimization flag: -O3
   Compare the timings with the result obtained under 1.

1-step: Compilation + linking:

gcc hello.c -o hello.x *(C source code)*

gfortran hello.f -o hello.x *(Fortran source code)*

2-step process:

Object compilation: **gcc -c hello.c** *(Creates hello.o)*

Linking: **gcc hello.o -o hello.x** *(Links hello.o with sys. libraries into an executable)*

Using optimization: **gcc -O3 hello.c -o helloFast.x**

Solutions to Exercise 1

1. Compiling a C program:

   1-step:

   gcc cpi_ser.c -o cpi_ser.x (Time: ~1.625 s)

   2-step:

   gcc –c cpi_ser.c

   gcc –o cpi_ser.x cpi_ser.o

2. Compiling a Fortran program:

   2-step:

   gfortran –c pi3_ser.f

   gfortran –o pi3_ser.x pi3_ser.o -- Errors (Missing dependencies)

3. Compiling the missing dependency + linking:

   gcc –c timer.c # (creates ctimer.o)

   gfortran ctimer.o pi3_ser.o –o pi3_ser.x

4. Compiling with –O3:

   gcc –O3 cpi_ser.c –o cpi_ser.fast.x

   or:

   gcc –c –O3 cpi_ser.c

   gcc –o cpi_ser.fast.x cpi_ser.o

Compiling program packages

- Some packages are far more complicated than one or two source files.
  - Many packages use gnu config/make
  - Others use cmake (useful for cross-platform)
  - There are other less common build tools
- You will almost certainly encounter a package like this if you continue in scientific computing
  - CHPC can help compile programs (can be hard) but knowing how to do it yourself is useful.

GNU config and make

- Configure: A scripting utility that checks for certain libraries and applications, as well as compiler capabilities, and creates makefiles.
  - Executed by the **./configure** script in the package directory.
  - You can use **./configure --prefix=<PATH>** to decide where to install the package, otherwise it will install in the same location as the package source.
- Make: Takes instructions from a makefile (a special script) to compile source in order to make a program.
  - As simple as executing **make** in a folder with a Makefile (or specifying the makefile with **-f**)
  - Sometime you need to use **make install** to finish the installation process.

Exercise 2

In this exercise you will download and compile the GSL: [GNU Scientific Library]() library.

1. Make a directory called `src/gsl` and cd to it.
2. Download and untar the gsl library:
```
wget ftp://ftp.gnu.org/gnu/gsl/gsl-2.4.tar.gz
tar  -zxvf gsl-2.4.tar.gz
cd gsl-2.4
```
3. Set gcc compiler flags (in bash):
```
export CC=gcc ;   export CFLAGS="-m64 -O2 -fPIC"
```
4. Configure and make the library in the src directory
```
./configure --prefix=$HOME/LinuxBuilding/pkg/gsl/2.4 --with-pic
make -j 2
```
5. Install
```
make install
```
6. Check to see if the library was installed properly in $HOME/LinuxBuilding/pkg/gsl/2.4/lib

Exercise 2 solution

```
mkdir -p $HOME/LinuxBuilding/src/gsl
cd $HOME/LinuxBuilding/src/gsl
wget ftp://ftp.gnu.org/gnu/gsl/gsl-2.4.tar.gz
tar  -zxvf gsl-2.4.tar.gz
cd gsl-2.4
export CC=gcc
export CFLAGS="-m64 -O2 -fPIC"
./configure --prefix=$HOME/LinuxBuilding/pkg/gsl/2.4 --with-pic
make -j 2
make install
```

Spack - easier alternative

- CHPC installs most programs with the Spack package manager
- Users can leverage CHPC installed packages for their builds
- See https://www.chpc.utah.edu/documentation/software/spack.php for details how to set up and use Spack
- ```
  module load spack
  spack spec -I gsl%gcc@8.5.0
  spack install gsl%gcc@8.5.0
  ```
  ```

What is a library?

- Library: collection of objects
- Can contain data sets, functions, classes, etc.
- Primary use: **reuse of the code**

  e.g. gsl
- There are 2 ways to build a library:
  - Static library: **.a** suffix
  - Dynamic library: **.so** suffix

Static Libraries

- Appeared first in time
- Have the **.a** suffix (**a**rchive file)

  e.g. libgsl**.a**, libz**.a**, etc.
- Specifics:

  -> copies the required objects from the library in the executable at linking time.

  -> larger executables

  -> requires more memory to load

  -> more portable & faster

Dynamic libraries

- Have the **.so** suffix (**s**hared **o**bject)

  e.g. libgsl**.so**, libz**.so**

- Specifics:

  -> no copy of object files into exe at linking

  -> require less disk space & less memory

  -> lib. can be updated without recompiling exe

  -> a little slower than static case

Create a library & use it.

- **Goal 1**:

  -> we want to create a 1D num. integ. library

  -> the library (**integ** directory) contains:

  src directory:

  a. *mc.c* (Monte-Carlo integration -> depends on gsl)

  b. *trap.c* (Trapezoid rule)

  include directory:

  integ.h (header file)

  lib directory:

  we will create **libinteg.a** & ***libinteg.so***

- **Goal 2**:

  Use **newly created libraries** to create executables.

Create the Static Library

- Step 1: Generate the object files

```
cd integ/src
gcc -c -I$GSL_ROOT/include -I../include mc.c
gcc -c -I../include trap.c
```

- Step 2: Create the static library libinteg.a

```
cd integ/lib
ar -crv libinteg.a ../src/{mc.o,trap.o}
```

A little more on ar(chive)

- *ar -t libinteg.a* # Lists/Tabulate content archive
- *ar -x libinteg.a mc.o* # Extract mc.o WITHOUT deletion in the archive
- *ar -d libinteg.a mc.o* # Delete mc.o from archive
- *ar -q libinteg.a mc.o* # Append mc.o to archive
- *ar -r libinteg.a mc.o* # Replace mc.o in archive
- *man ar*

Create a Dynamic Library

- Step 1: Generate the object files (use **-fPIC** compil. flag -> to avoid linking error)

```
cd integ/src
gcc -c -fPIC -I$GSL_ROOT/include -I../include mc.c
gcc -c -fPIC -I../include trap.c
```

- Step 2: Create the dynamic library libinteg.so

```
cd integ/lib
gcc -shared -fPIC -o libinteg.so ../src/{mc.o,trap.o}
```

A few useful commands/tools

- ldd [options] file

  find a *program*'s/library's shared libraries

  (ldd: list dynamic dependencies)

```
[u0253283@dirac:lib]$ ldd libinteg.a
 ldd: warning: you do not have execution
permission for
     `./libinteg.a'
 not a dynamic executable
```

```
[u0253283@dirac:lib]$ ldd libinteg.so
linux-vdso.so.1 =>  (0x00007fffc999b000)
libc.so.6 => /lib64/libc.so.6 (0x00002ae6d803d000)
/lib64/ld-linux-x86-64.so.2 (0x000055c2bddc4000)
```

NEVER use ldd against untrusted code (-> will be executed!)

-> use objdump instead

```
u0253283@dirac:lib]$ ldd -v libinteg.so                   #
Verbose output => GLIBC
    linux-vdso.so.1 =>  (0x00007ffd66df1000)
    libc.so.6 => /lib64/libc.so.6 (0x00002b16e5789000)
    /lib64/ld-linux-x86-64.so.2 (0x000055be9ac18000)

    Version information:
    ./libinteg.so:
        libc.so.6 (GLIBC_2.2.5) => /lib64/libc.so.6
    /lib64/libc.so.6:
        ld-linux-x86-64.so.2 (GLIBC_2.3) => /lib64/ld-
linux-x86-64.so.2
        ld-linux-x86-64.so.2 (GLIBC_PRIVATE) => /lib64/ld-
linux-x86-64.so.2


[u0253283@dirac:lib]$ ldd -d libinteg.so    # Reports
missing objects
undefined symbol: gsl_rng_default    (./libinteg.so)
    linux-vdso.so.1 =>  (0x00007ffde5318000)
    libc.so.6 => /lib64/libc.so.6 (0x00002ba359c5b000)
    /lib64/ld-linux-x86-64.so.2 (0x0000555f96ad9000)
```

- nm [options] file

  prints the **n**ame **l**ist (i.e. symbol table) of an object file

  Default output:

  1. Virtual address of the symbol
  2. Character/Symbol type:

     lower case: local   upper case: external

     A/a: Global/local abs. type   (Not changed when linking)

     B/b: Global/local uninitialized data

     D/d: Global/local initialized data

     f: Source file name symbol

     …

     L/l: Global/static thread-local symbol

     T/t: Global/local text symbol

     U: Undefined symbol
  3. Name of the symbol

- Note on the nm flags/options:

  -> nm --help : list an overview (you can also use man nm)

  -> nm -u file : list only the undefined symbols

  undefined: can either be **unresolved or can be resolved at runtime** through shared libraries

- objdump [options] file

provides thorough information on object files

```
 # Contents of the file header
 [u0253283@dirac:mytest]$ objdump -f main_d1

main_d1:     file format elf64-x86-64
architecture: i386:x86-64, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x00000000004007a0


# Dumps assembler of the executable content
[u0253283@dirac:mytest]$ objdump -d main_d1 | less
main_d1:     file format elf64-x86-64


Disassembly of section .init:

00000000004006e0 <_init>:
4006e0:       48 83 ec 08                   sub    $0x8,%rsp
4006e4:       48 8b 05 0d 19 20
00      mov    0x20190d(%rip),%rax        # 601ff8     <_DYNAMIC+0x210>
4006eb:       48 85 c0                      test   %rax,%rax
4006ee:       74 05                         je     4006f5 <_init+0x15>
4006f0:       e8 4b 00 00 00                callq  400740
<__gmon_start__@plt>
4006f5:       48 83 c4 08                   add    $0x8,%rsp
4006f9:       c3                            retq
```

To find dependencies

```
[u0253283@dirac:mytest]$ objdump -p main_d1 | grep NEEDED
  NEEDED                     libgslcblas.so.0
  NEEDED                     libgsl.so.23
  NEEDED                     libm.so.6
  NEEDED                     libinteg.so
  NEEDED                     libc.so.6
```

Exercise 3

- Browse through the files within:

  $HOME/LinuxLibs/integ/{src,include}

- Generate the **static** library ***libinteg.a*** in the directory

  $HOME/LinuxLibs/integ/lib

- Use the **ar** command to see the content of *libinteg.a*
- Check the following commands:

  nm mc.o

  objdump –f –s –d mc.o

Exercise 4

- Generate the **dynamic** library ***libinteg.so*** in the directory

  $HOME/LinuxLibs/integ/lib

  NOTE:

  To generate the dynamic library you MUST compile the source files with the –fPIC flag!

- Check the libraries using the ***ldd*** command:

  ldd ./libinteg.so

  ldd ./libinteg.a

The linking process => exe

- How to do linking?

  *gcc -o name_exe *.o* [library_info]

- [library_info]

  - If you use a library such as libm.{so,a} (sqrt, exp,…)

    => ‘ -lm ‘ is sufficient

  (no need to specify directory where libm.{so,a} is stored)

  Why? -> /etc/ld.so.conf.d directory

  ldconfig -p | grep libm

  - Otherwise (if library libmylib.{so,a} can’t be found **during linking**)

    => ‘-L$LIBDIR -lmylib ‘

  LIBDIR: directory where libmylib.{so,a} is stored

- Note:

  a. If the dyn. & static version of the same lib. are both present in LIBDIR=> dyn. library will be taken.

  b. If you want the static library to be taken, then use -L$LIBDIR $LIBDIR/libmylib.a

- Example:

  *gcc -o main_d1 main.c functions.c -L$INTEG_ROOT/lib -linteg *
  -L$GSL_ROOT/lib -lgslcblas -lgsl -I$INTEG_ROOT/include -I. -lm

  1. Dynamic libraries will be taken
  2. Error will be thrown when trying to run ./main_d1 => Why?

```
[u0253283@dirac:mytest]$ ./main_d1
./main_d1: error while loading shared libraries:
libgslcblas.so.0: cannot open shared object file: No such
file or directory

[u0253283@dirac:mytest]$ ldd main_d1
    linux-vdso.so.1 =>  (0x00007ffc551b6000)
    libgslcblas.so.0 => not found
    libgsl.so.23 => not found
    libinteg.so => not found
    libm.so.6 => /lib64/libm.so.6 (0x00002b7549f21000)
    libc.so.6 => /lib64/libc.so.6 (0x00002b754a224000)
    /lib64/ld-linux-x86-64.so.2 (0x0000558b57cb2000)
```

Executables

- Two types:

  A. Static executable

  B. Executable relying on dynamic libraries

- A. Static executable:

  -> You **MUST** use **STATIC** libraries

  -> Use the '-static' flag (GNU) at linking time

  -> Example:

```
gcc -static -o main_s2 main.o functions.o \
-L$INTEG_ROOT/lib $INTEG_ROOT/lib/libinteg.a \
-L$GSL_ROOT/lib $GSL_ROOT/lib/libgsl.a \
$GSL_ROOT/lib/libgslcblas.a -lm
```

- B. Exe based on dynamic libraries

  The executable **MUST** find the dyn. libraries at **RUNTIME**

  (remember: "*error while loading shared libraries* …")

  Option 1: *ldconfig* command

  Command to create & maintain the cache for dyn. libraries (sys. admin tool => No Option for users!)

```
[u0253283@dirac:mytest]$ ldconfig -p | grep gsl
libgslcblas.so.0 (libc6,x86-64) => /lib64/libgslcblas.so.0
libgsl.so.0 (libc6,x86-64) => /lib64/libgsl.so.0

[u0253283@dirac:mytest]$ ldconfig -p | grep libinteg
[u0253283@dirac:mytest]$
```

Option2:

If the correct version of the library is **NOT** in the ldconfig cache, the user needs to supply the lib. info to the exe.

a. At **Runtime: -> use the LD_LIBRARY_PATH env. var.**

```
[u0253283@dirac:mytest]$ ldd ./main_d1
  linux-vdso.so.1 =>  (0x00007ffc3fc96000)
  libgslcblas.so.0 => /lib64/libgslcblas.so.0
(0x00002aab37273000)
  libgsl.so.23 => not found
  libinteg.so => not found
  libm.so.6 => /lib64/libm.so.6
(0x00002aab374b1000)
  libc.so.6 => /lib64/libc.so.6
(0x00002aab377b3000)
  /lib64/ld-linux-x86-64.so.2 (0x000055bc8aeba000)
```

Solution:

`export LD_LIBRARY_PATH=$LIBDIR:$LD_LIBRARY_PATH` (Bash shell)

`setenv LD_LIBRARY_PATH $LIBDIR:$LD_LIBRARY_PATH` (Tcsh Shell)

=> the "not found " message will disappear

b. At **Linking Time**:

Use the following construct when linking the code:

“ -Wl,-rpath=$LIBDIR -L$LIBDIR -lmylib “

Example:

```
gcc -o main_d2 main.o functions.o \
-Wl,-rpath=$GSL_ROOT/lib -L$GSL_ROOT/lib -lgslcblas -lgsl \
-Wl,-rpath=$INTEG_ROOT/lib -L$INTEG_ROOT/lib -linteg -lm
```

Exercise 5

- We will now create executables based on the *gsl* and *integ* libraries.
- Create executables (within mytest) in different ways:

a. main_d1: using **only dynamic libraries** (gsl and integ)

without using the –Wl,-rpath construct

b. main_d2: using **only the dyn. libraries** (gsl & integ)

but **use the –Wl,-rpath construct**

c. main_s1: use the **dyn. gsl libraries** (**using the –Wl,-rpath construct**)

but use the **static library libinteg.a**

d. main_s2: create a **completely STATIC** executable.

(This requires glibc-static.x86_64 to be installed on the machine)

Questions?

Email helpdesk@chpc.utah.edu